编程狂人

Programming Madman

NO.2

推酷

关于推酷

推酷是专注于 IT 圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等多方面内容，满足你日常的专业阅读需要。我们针对 IT 人还做了个活动频道，它聚合了 IT 圈最新最全的线上线下活动，使 IT 人能更方便地找到感兴趣的活动信息。

关于周刊

推酷周刊是专为 IT 人打造的行业技术周刊，目前推出的《编程狂人》是献给广大的程序员们。 我们利用技术挖掘出那些高质量的文章，并通过人工加以筛选整理出来。 每期的周刊一般会在周六到周一的某个时间点发布。关于《编程狂人》的名号还有个小插曲， 因为我们一直很崇尚程序员这个群体，所以我们一开始将周刊定名为《编程匠人》， 不过做封面设计的朋友不知怎么就改成了《编程狂人》，我们想想也还不错，就使用了。关于《编程狂人》， 如果你有意见和建议，欢迎反馈给我们。

联系我们


tuicool2012


164644910

推酷网

下载 APP

Android版本

立即下载 Android

iPhone版本

免费下载就在 App Store

目录

业界新闻

用数据分析 AV 女优，寻找下一位苍井空

JavaScript 是最受欢迎的远程办公编程语言

JVM 垃圾收集器使用调查：CMS 最受欢迎

GitHub 创始人 Preston-Werner 专访：细说 GitHub 成长史

Atlas 2.0.0 发布，来自 360 的 MySQL 中间层

百度开源平台上线，聚合百度开源项目

前端开发

Kraken：改变 PayPal 开发文化的 Node.js 框架

JavaScript 核心

想学响应式设计？来看史上最全的响应式设计资源库

前端性能优化的 14 个规则

编程语言

一个 Bump Pointer Allocator

Python 项目自动化部署最佳实践@搜狐

Java SE 8：标准库增强

Erlang 数据类型的表示和实现（4）——boxed 对象

精简自己 20%的代码

Android 应用性能分析

Rails 3.2 性能： 更慢了?

目录

后端架构

《基于 Oracle 的 SQL 优化》作者谈 SQL 优化的重要性

图灵访谈 ： 世界级 Oracle 专家 Jonathan Lewis：我很为 DBA 们的未来担心

映射的存储模型 – 面向列的存储和行存储

使用 MySQL 自身复制来恢复 binlog

自动摘要算法

关于 MySQL count(distinct) 逻辑的另一个 bug

有关于存储过程的一个笑话

程序人生

揭开程序员装 13 行为的面具

6 家科技公司 CEO 分享第一份工作经验

程序员的敌人

技术人创业可能面临的挑战

用数据分析 AV 女优，寻找下一位苍井空

2008 年，美国国家工程学院对 25000 名工程师进行调查，并借此总结出一份他们心目中 21 世纪将要面临的宏大挑战，这是预测，同时也是创新的根源推动力。在排名前 14 名的挑战中，例如“先进的个性化学习”“制造用于科学探索的工具”“增强虚拟现实”……等等已经在过去的五年里得到了飞速的发展。

当年，因为诸葛亮对天气的预测，让他成功的借来东风，又凭借对周瑜的人性预测，让他成功的逃脱魔爪，由此可见合理有效大胆的预测是一件利国利民功在千秋的事情。

提升逼格的套话已经说完，那么，让我们开始从 AV 女优群体的数据分析中，预测谁将成为下一个苍井空老师，毕竟她已经 30 岁了。

老兵永远不死，只会慢慢凋零……

一.数说 AV 女优群体

1.数据样本

两个月前，我曾写过一篇《大数据，你能在色情行业做什么》，但是里面的数据样本和分析主要是针对欧美艳星，因为欧美的从业者们拥有一个非常完善的信息资料库，从个人常规资料到身体哪个地方有纹身，均有数据可查可分析。欧美很容易把所有的行业都做的信息化科技范儿十足，例如《生活大爆炸》里，那用火星机器人给自己撸管的霍华德。

但是不少朋友在看了那篇文章后,纷纷发来热情洋溢的贺电,请求我能走出国门放眼小小寰球,分析一下与我们一衣带水的日本 AV 女优市场。与欧美同行相比，东瀛 AV 市场整体就显得更娱乐化一些，虽然经常有影片的创意让人瞠目结舌，但是缺少整体的通盘筹划和信息化建设。

特别需要一提的是，很多 AV 事务所官网或者是成人网站，是禁止海外 IP 浏览

的。在一个初冬阳光明媚的清晨，暖洋洋的日光撒在我的身上，我在 QQ 上点开一个人的头像，默默地发去一条链接。

过了两分钟，远在澳大利亚的他恨不得哭了，“独在异乡为异客，每逢佳节倍思亲，一骑红尘妃子笑，故人发个黄网来。”

他说他在那边很孤单，很久没有跟人聊天了，没想到我会突然跟他说话，但是发一条打不开的成人网站链接就不对了。这时我的心理平衡了，原来日本成人网站不光屏蔽发展中国家的屌丝，连发达国家也一起屏蔽啊！

在这种无比艰难的情况下，汇总了 1129 名 AV 女优的资料作为数据样本，选择的主体基本是单体女优和企划单体女优，要求是至少出过一部自己为独立女主的影片，这些人是 AV 女优市场的主力军。而那些路人甲一般的企划女优实在如海洋般浩瀚，非本人能力范围内。

2.基本信息

从样本中所有 AV 女优的星座来说，排名第一的是射手座，占总人数的 12%，紧随其后的是双鱼和白羊座，这三个星座所占比重均超过了 10%。（av 女优的星座排名：1 射手座 2 白羊座 3 双鱼座 4 处女座 5 天蝎座）

处女座虽然和天蝎座一起并列第四，但是值得一提的是，在已知的自杀 AV 女优

中，处女座占了半壁江山，例如 2007 年在家中上吊的美咲沙耶，自杀前患有严重的抑郁症和酗酒。而人数最少的星座则是金牛座，恐怕是因为 AV 女优这个职业与内向沉稳的金牛座原本就气场不合。

而在血型方面，居然 40.8%的女优是 A 型血。顺手搜了一下 A 型血人的特点，“能确实遵守约定和规则”“无论如何也会努力达成目标，耐劳力很强”，看到这里，我真的相信了血型性格说，毕竟对于 AV 女优来说，在录像带时代一部片子 40-60 分钟，但是到了 DVD 时代，基本是 90 分钟起，长的也有 3-4 个小时的。如果没有 A 型血坚忍不拔的性格，那么哪怕自己身体不动，任人蹂躏 90 分钟也是一大挑战能消耗很多卡路里。其次是 O 型血，也占据了 31%，而 B 型和 AB 型相比之下，就少得可怜了。

对于大部分的女优，她们所使用的名字都是艺名，由此也可以猜到她们资料中的出生地恐怕也真实不到哪里去。事实正是如此，在 769 个能查到出生地的女优中，有 30%的人说自己是东京人，其次有 10.6%的人说自己是神奈川县的。如果我们借此就粗暴的断定东京盛产 AV 女优是不公平的，这就跟市场上其实没有那么多阳澄湖大闸蟹一个道理。

3.身高身材

从一个时代人平均身高的变化，可以看出一个民族的营养状况。而从 AV 女优的身高变化，可以看出多年来人们选美的标准变化。

根据出道时间来区分 AV 女优，在 90 年代出道的女优平均身高为 157.8 厘米，而到了 2000-2004 年，这五年间出道的 AV 女优平均身高已经到了 158.5 厘米，随着时代的发展，AV 女优的身高越来越高，2010-2013 这四年出道的新人，平均身高已经突破了 160 大关，达到了 160.42 厘米。

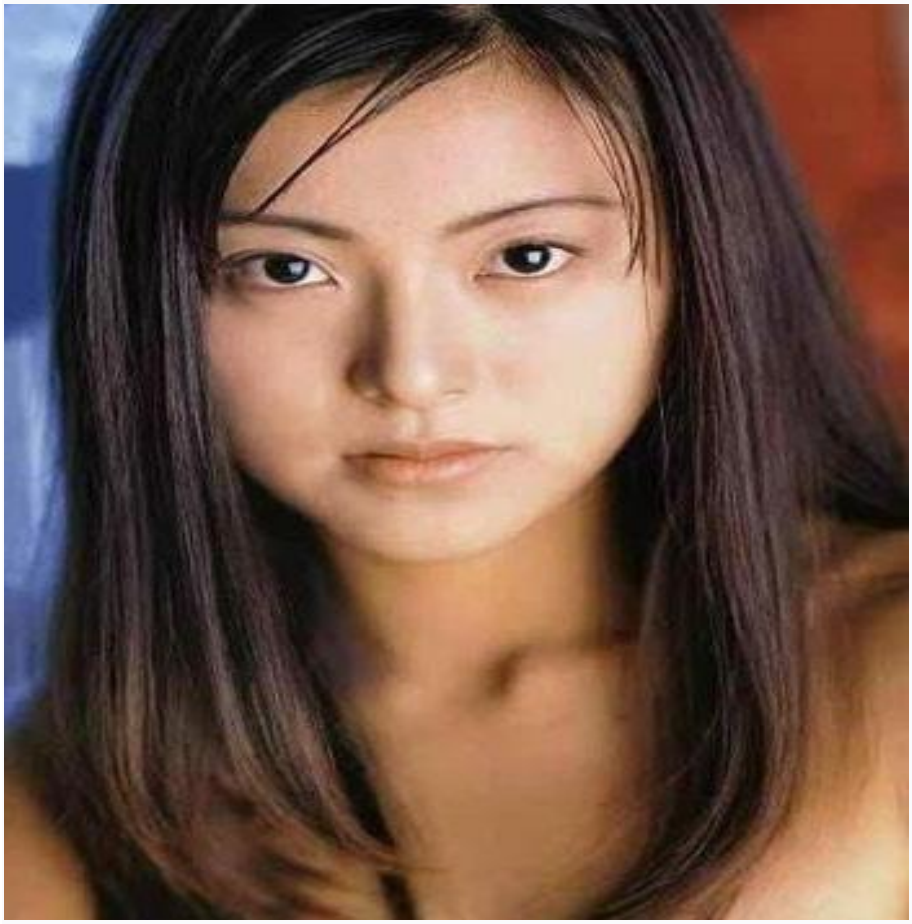
娇小可爱一直是日本女性的代名词，而 AV 女优中萝莉系女生也颇多，凭借着姣好的面容和娇小的身材让大叔和屌丝们泛起浓浓的保护欲望。

在 1129 名 AV 女优中，身高低于 150 厘米的有 40 人。但是常言道“有志不在年高，有胸不在身高”，这 40 人中，排名第一的居然是 E 罩杯，有 10 个人拥有这傲人的胸怀。另外 G 奶就有五人，另外还有身高 148 厘米却有着 J 奶的芹泽由衣。

这其中值得一提的是长濑爱，作为当年和武藤兰齐名的行业翘楚，赖以成名的绝技是妖媚一般的腰部扭动，但是正因为她的职业操守，导致了她因为腰椎间盘突出而短暂停止演出。而她的身高，也因为腰椎间盘突出，由 145 厘米变成了 143 厘米。

看到这些感人的事迹，我简直都要哭了。一个身高仅仅 145 厘米的小女孩，对工作的严谨与敬业让她早早的患上了职业病，甚至她曾经在拍摄中，因为运动量过大而导致休克，紧急送去医院。**我掩卷深思，不知道她们的工伤医疗报销，单**

位负担多少比例。



而身高在 150-159 厘米区间内的 AV 女优有 542 人，占据总数的 48%。而其中 A 罩杯的有 4 人。要知道在这个群体中，如果上帝没有给你一个高耸的胸部，也没有给你修长的美腿，那么一定会在其他地方给你补偿。例如其中小泽圆，身材并不突出，但是凭借自己清新可人的容貌拍摄了上百部 AV 影片，也参演了香港著名的三级片《偷情男女》，卖力的演出赢得了观众的一致好评。因为她作品的屡破销量，被人称为“AV 届的女皇”。(她的罩杯各处说法不一，有 A 有 B，但是 83CM 的胸围至少证明了不大。)

这个事情告诉我们，如果看官你或者矮或者穷或者丑或者挫或者傻，这都不可怕，

因为上帝是公平的，你会在其他地方体现自己的价值。如果你又矮又穷又丑又挫又笨，那么我只好说，你就是上帝用来激励成功人士的参照物。

在此区间内，最大比重的是 C 罩杯，占据了总数的 21%，这是一个平胸分界线。而排名第二的是 E 罩杯，占了 16%。G 罩杯以 9%排名第五，其中最知名的当属身高 155 厘米胸围 90G 的苍井空老师。

而位于金字塔顶端的，是 S1 旗下的专属 AV 风子，她虽然没有冷艳的容颜，152 的身高也容易消失在人群中，但是她那傲人的 P 罩杯却成为其独特的标识。而福岛地震后，她又出来为福岛的西瓜背书，证明其天然无污染，简直就是新时代德艺双磬的典范，就是长得差点儿意思。

在近年来,虽然巨乳一词出现频率最高,但是”长身”一词也异军突起。毕竟对于看多了娇小可人的小女生，突然来一高个美腿 MM，别有一番风味。身高最高的当属星野沙果恵，身高 1 米 88。

最出名的当属号称日本女排国手内田真由，身高 1 米 82。不过遍查 07-09 年的日本女排阵容，至少在国际大赛上并没有这个 182 的姑娘。哪怕入选过国家集训队，这个噱头依然足够拍摄商做宣传了。她的片子适合重口味人士以及想换性取向的人士观看，基本上是男优成了被欺凌被蹂躏被侮辱的一方，让人不得不唏嘘感叹，身高才是硬通货啊。

4.年龄

时代在发展，大家的营养越来越好，从身高到罩杯均有了长足的进步，例如 70 后女优一共有 64 人，平均身高为 157.8 厘米，A-C 罩杯占据了 42%。而 80 后的平均身高为 158.2 厘米，A-C 罩杯的比重剧降到了 23.3%，可见如果你低于 C 杯，怎么好意思在这行里混呢?

而 90 后女优的平均身高已经到了 159 厘米，罩杯则被清一色的“童颜巨乳”所占领，仅仅是从 G 杯到 M 杯的巨乳女优，就占了 32%，代表人物当属异常火爆的橘梨纱，从 AKB48 成员下海拍 AV，纯情冷艳的面容成为宅男女神，从 2 月其第一本全裸写真发售到八月的《橘梨纱紧缚中出し》，她的每一部作品都成功

横扫 AV 界，在 DMM 网站的单一作品 2013 年上半年销量排行榜上，前三名全被她占据。为了搏出位，同为 90 后美少女的上原亚衣甚至不惜与黑人大战，但是仍然不敌脱衣服就能脱 30 分钟的橘梨纱，这就是命。

5.作品数目

有句话叫做“以大多数人的努力程度之低，根本没有轮到去拼天赋”。这句话用在 AV 女优行业再合适不过。要想出人头地，除了爹妈给的硬件，更需要努力拼搏的精神。

例如武藤兰老师，在 2002 年一共有 212 部 AV 作品面世，并因此创造了“一年之内售出最多色情影片女演员”的吉尼斯世界纪录，而 2003 年，她勇攀高峰，超越自己，创纪录的出演了 304 部作品，成为年度最多作品纪录保持者。365 天演了 304 部作品，她就是传奇。

而 1997 年出道的风间由美，则以当时少见的 G 罩杯 19 岁少女成为了宠儿。岁月如梭，转眼间，在去年 10 月，风间由美举办了自己从业 15 周年的纪念活动，在 15 年如一日的辛勤劳动中，她以 1724 部作品成为了现役 AV 女优中的敬业模范。

排名第二的花蕾，虽然 2006 年才出道，但是作品数已经达到了 1524 部，87 年出生的她如果能保持这个热血狂飙的速度，那么追上风间阿姨指日可待。

6.出道原因等奇葩数据

虽然我们很愿意相信，这些失足少女投身风月场合都是因为有着一个悲惨的童年，要供应弟弟妹妹上学，而不得不毅然决然的脱掉衣裳，眼含热泪的躺倒在榻榻米上。

我国淫民日报的网站曾经发过一条新闻，叫做《日本 AV 女优背后的病态社会》，在里面描绘了 AV 女优因为要承担起家庭的重任而走上了拍摄 AV 的道路。

但是从已知的数据来看，成名快收入高成为了新出道女优的首选原因，排名第三的是“刺激，体验不一样的人生”。例如 1994 年出生的仓多真央，她出道的原因据称是多次给 AV 制造商 SI 公司写信求援，说男友无法让她满足。后来，SI 就把她打造成了女优。

而在 1129 名女优中，以处女身份出道的共有 7 人，其中椎名彩为了出位，不惜在荧幕前献出了自己的处子之身，只可惜销量巨惨，只拍了四个月就退出了江湖。

但是几个月之后，她再次重出江湖，不过换了一个名字，叫做水树美叶。

对于大神一条绮美香，嗯，其实大神也可以叫做“大婶”，作为 48 岁才出道的她，我实在无法再漠视世道的艰辛和人心的不古。而如果有重口味的有心人看了视频，再回过头来看一下她的封面，一定会哭着对我说，封面都是骗人的。

二.谁会是下一个苍老师

虽然苍老师在日本本土的影响力已经在走下坡路，但是最近两年，其在中国内地市场掀起了一波又一波的 AV 女优来华潮。从苍井空到波多野结衣再到冲田杏梨，从麻生希到泷泽萝拉再到松岛枫，这些硬盘女神一个个的走到了广大宅男的身边。

那么，谁会像苍老师一样，成为下一个全民狂欢的偶像呢?

12 月 1 日，日本 AV 界最著名的“2013 日本成人放送大赏”将开启投票，这一

奖项是由主办方与 16 个成人频道一起从日本当年上市的众多成人影片中，选出入围名单，再开启网友投票，从而选出当年度最佳女优，最佳新人女优和最佳熟女女优，是日本 AV 领域的最高奖，堪比奥斯卡颁奖。顺便说一句，为了体现演员与观众间的鱼水之情，获奖者最后上台后，需要向台下扔出自己的亲笔签名小内裤，以感谢热心观众的支持。

如果把日本成人放送大赏之前五年的最佳女优，DMM 网站上的 2013 年上半年 AV 销量前三名，新浪微博粉丝数最多的两人分别选出来，一共 10 人。这 10 人无疑涵盖了中日两国人民共同的审美观,对这 10 人进行一次人物肖像合成工作，至少能管中窥豹看出什么样的女优最有可能成为下一个 AV 女神。

日本成人放送大赏过去五年的最佳女优分别是佐藤遥希,成濑心美,麻仓优，原纱央莉，明日花绮罗。2013 年 DMM 网站统计的上半年销量前三名分别是橘梨纱，上原亚衣，彩美旬果。新浪微博 AV 女优粉丝数最多的五个人为苍井空，波多野结衣。

当用图像合成软件，进行两两捉对厮杀之后，就能得到的一张 AV 女优大众脸。

按照著名美剧<<犯罪心理>>里的专业描述,我们对未来最有可能成为下一个 AV 女神的女子进行一下人物侧写。

“她家境比较富裕，并不是苦大仇深卖身葬父，射手座 A 型血，从小喜欢古典

芭蕾和骑马，性格外向渴望尝试不同的生活，追求刺激和成就感。身高在 165 厘米左右，E 罩杯，长相如下图所示。”

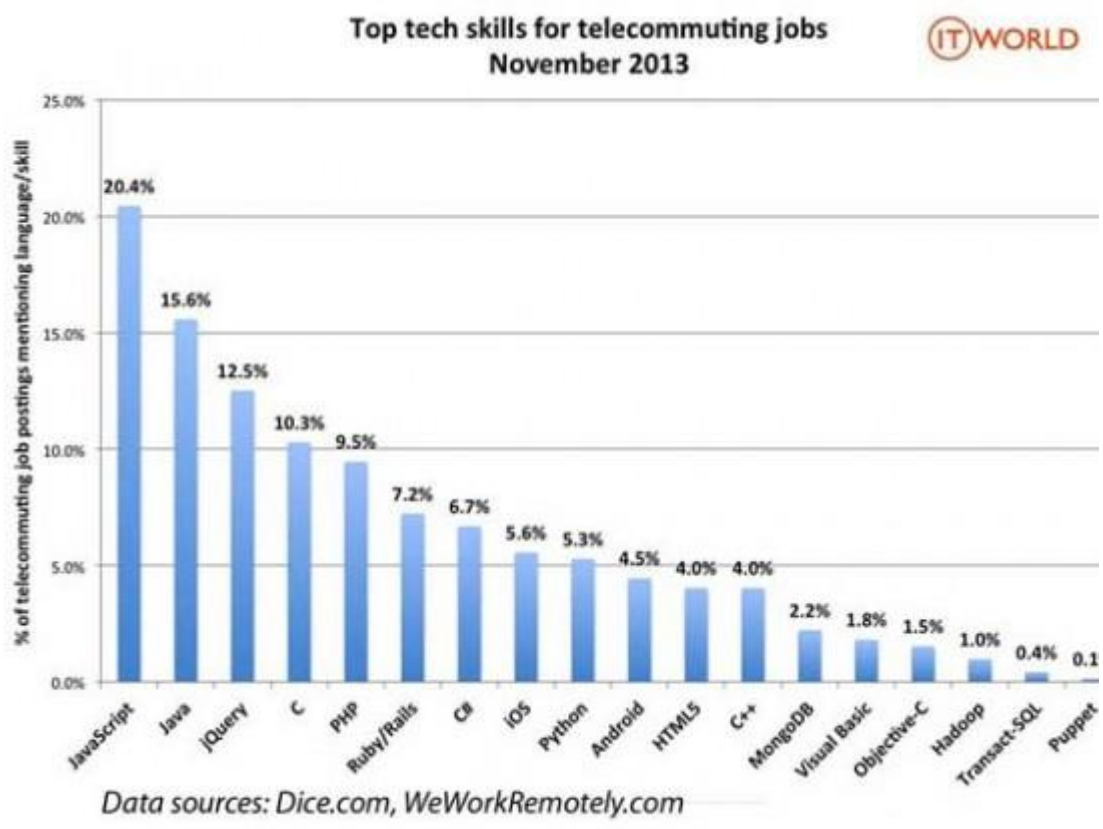
三.尾声

如果各位同学真的在马路上见到一位这样的姑娘，请不要贸然的上去搭讪，不要急于挑战这类似哥德巴赫猜想一般的难题，因为这样的姑娘，不是普通人可以驾驭的。

我们唯一能做的，就是放下工作的压力，忘记生活的烦恼，静静的双手合十，用自己一颗赤诚的心祝愿她早日成为我们大家的硬盘女神。

原文 http://www.huxiu.com/article/23735/1.html?utm_source=Tuicool_Weekly

JavaScript 是最受欢迎的远程办公编程语言



对于那些喜欢在家办公的软件开发人员、web 设计师等自由职业者来说，学习哪些语言最有利于找到一份可以穿着睡衣抱着狗狗喝着咖啡打电脑的工作呢?

近日 ITworld 的编辑 Phil Johnson 进行了一番研究发现：老牌语言 JavaScript 有可能是远程办公者的最佳选择。Johnson 的调研方法很简单，上各大 IT 招聘网站搜索职位，通过职位说明中各编程语言的出现频率来判断其受雇主的欢迎程度，统计结果如上图（编者按：出现频率高未必薪酬高，通常 Ruby 和 Python 开发者的报酬要比 PHP 和 JS 高很多）

Johnson 的调查方法如下：

在 Dice 和 We Work Remotely 上搜索 719 个远程技术工作；在 Dice 上查找所有编程工作区域的职位（141 个），在 we work remotely 上在“高级搜索页面”将搜索条件限定为“远程办公”（578 个），通过 JavaScript 等不

同的关键词搜索职位。

参考了 TIOBE 编程社区指数（2013 年 11 月）的十大编程语言：C、Java、Object-C、C++、C#、PHP、Visual Basic、Python、Transact-SQL、JavaScript。

同时参考了 Indeed.com 上的最热工作技能，分别是：HTML5,MongoDB、iOS、Android、Puppet、Hadoop、JQuery，其他如移动应用开发、PaaS、社交媒体等过于范范的技能描述被忽略。

从 Johnson 的分析图表可以看出，JavaScript 在提及频率上名列榜首，可见随着互联网的普及这项技能的需求越来越多，虽然 JavaScript 在 TIOBE 编程语言指数中仅仅排名第十，但毫无疑问 JavaScript 已经是最为热门的编程语言。

总的来说，web 开发使用的语言和工具是目前远程办公劳务市场的热点，例如 JavaScript、jQuery（第三名，12.5%）和 PHP（第五名 9.5%），iOS（第八名，5.6%），Android（第十名，4.5%）和 HTML5（第十一名，11.4%）

当然，学习经典语言是没有错的，Java（第二名，15%）和 C（第四名，10%）依然宝刀不老。

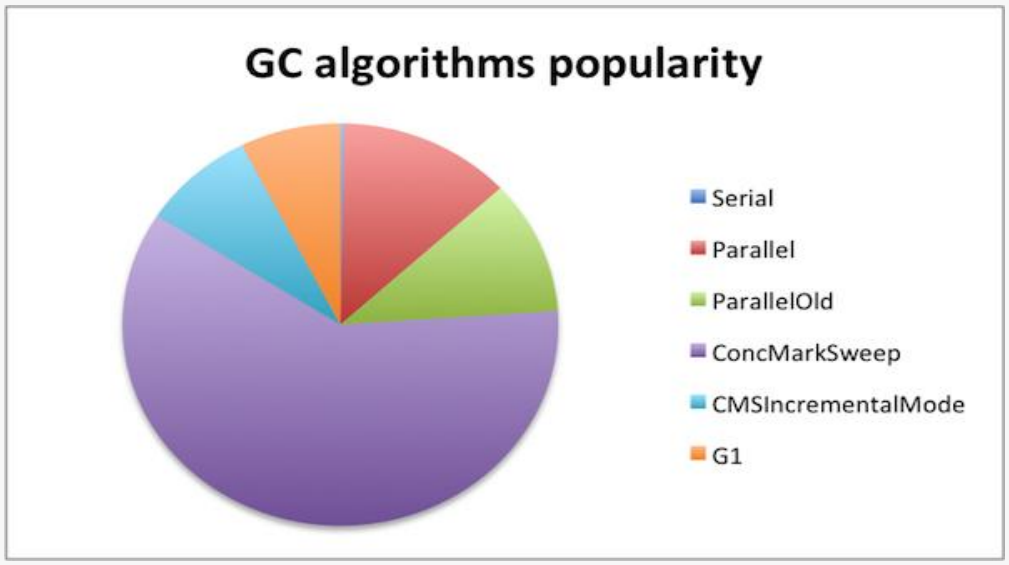
总之，Johnson 的这份榜单告诉那些自由职业者，如果你安于抱着猫猫狗狗穿着睡衣，过着报酬不太高但很惬意的远程办公生活，那么赶紧学点 JavaScript 吧。

原文 http://www.ctocio.com/ccnews/14115.html?utm_source=Tuicool_Weekly

JVM 垃圾收集器使用调查：CMS 最受欢迎

近日，Plumbr 公司对特定垃圾收集器（GC）使用情况进行了一次调查研究。本次研究的数据来自代表 2670 个不同使用环境的 84936 个案例。其中，13%的

环境已经明确指定了一个垃圾收集器，其余的根据 JVM 而定。在指定了明确垃圾收集器的 11062 个案例中，根据每个垃圾收集器使用的统计次数，研究人员做出了下面的垃圾收集器饼图：



GC 使用统计

名词解释

- Serial：串行收集器，当进行垃圾收集时，会暂停所有线程
- Parallel：并行收集器，是串行收集器的多线程版本，多 CPU 下
- ParallelOld：老年代的 Parallel 版本
- ConcMarkSweep：简称 CMS，是并发收集器，将部分操作与用户线程并发执行
- CMSIncrementalMode：CMS 收集器变种，属增量式垃圾收集器，在并发标记和并发清理时交替运行垃圾收集器和用户线程

- G1：面向服务器端应用的垃圾收集器，计划未来替代 CMS 收集器

87%的案例没有指定垃圾收集器

在解释垃圾收集器使用情况的详情之前，我们先看下其他 87%的案例为什么没有出现在上面的饼图中。从研究结果来看，有 2 个不同的原因导致了该情况的出现：

- JVM 对于默认情况的处理十分合理，开发人员无需指定垃圾收集器
- 对部分团队来说，程序性能可能优先级不高，致使没有指定垃圾收集器

所以，研究团队没有采用使用默认垃圾收集器的 JVM 案例。话又说回来，默认的垃圾收集器又是什么呢？这个问题既简单又复杂。如果你运行在 JVM 的客户端模式（Client）下，JVM 默认垃圾收集器是串行垃圾收集器（Serial GC，-XX:+USeSerialGC）；在 JVM 服务器模式（Server）下默认垃圾收集器是并行垃圾收集器（Parallel GC，-XX:+UseParallelGC）。至于是运行在 JVM 的客户端模式还是服务器模式，取决于下面情况：

Architecture	CPU/RAM	OS	D
i586	Any	MS Windows	C
AMD64	Any	Any	S
64-bit SPARC	Any	Solaris	S
32-bit SPARC	2+ cores & > 2GB RAM	Solaris	S
32-bit SPARC	1 core or < 2GB RAM	Solaris	C
i568	2+ cores & > 2GB RAM	Linux or Solaris	S
i568	1 core or < 2GB RAM	Linux or Solaris	C

JVM 客户端/服务器模式

大多数案例没有做出最佳选择

让我们回到已经明确指定垃圾收集器的 13%的案例，但仅有一小部分用户的决策是按照上述表格中的建议进行的。据统计，只有 31 个案例根据自己的机器性能选择了最佳的串行垃圾收集器，考虑到当前服务大多运行在多核服务器上，这个可以理解。

GC	Count
Serial	31
Parallel	1,422
ParallelOld	1,193
ConcMarkSweep	6,655
CMSIncrementalMode	935
G1	826

垃圾收集器使用类型统计

我们从上图可以看出，并行（Parallel）和 ParallelOld 使用次数很接近。如果觉得并行模式这一新生代收集器更符合你的需求，那就选择它。从第一张表格中我们也可以看出，并行垃圾收集器(Parallel)已经是大多数平台的默认选择。从这个方面讲，如果没有指定明确的垃圾收集器，也并不意味着默认使用的垃圾收集器不流行。

说到 CMSIncrementalMode 的使用情况，只有 935 个环境使用了该种垃圾收集器，相比而言，经典的 CMS（ConcMarkSweep）则有 6655 个环境使用了它。这里提示下大家，在并发阶段，垃圾收集器线程会使用一个或多个处理器。

增量式垃圾收集器是通过一定的回收算法，把一个长时间的中断，划分为很多个小的中断，以减少垃圾收集器对用户程序的影响。

研究中还有一个结果就是 G1 的采用率，有 826 个环境使用了该种垃圾收集器。但同等条件来讲，G1 比 CMS 性能表现会差一些。

以上就是本次研究的结果，希望对各位有用。

原文

http://www.iteye.com/news/28511-study-about-jvm-garbage-collector?utm_source=Tuicool_Weekly

创始人 Preston-Werner 细说 GitHub 成长史

摘要：从 2008 年创立至今，Github 已经完成了它的华丽转身，从一个开发者社区变成了一个免费开源代码托管平台。创始人 Preston-Werner 近日接受媒体采访表示，GitHub 未来希望扩展到教育和科学领域，所以也在积极与学校接触。

从 2008 年创立之初到现在，Github 已经完成了它的华丽转身，从一个开发者社区变成了一个免费开源代码托管平台。伴随社区的风靡，其创始人 Preston-Werner 也为越来越多人所关注，本文是对他的专访。

RW：和许多成功开发者一样，您也中途辍学，那你觉得对一个科技从业者的人来说，有必要上大学吗?

TP--W：这得分人，每个人从大学里学获得的东西都不同。

我上了两年大学，这两年完全改变了我的生活。如果不上大学，可能完全是另外一个样子。大二那年夏天，我在一家做 Java 开发的创业公司工作，那里工作氛围特别好，会觉得和整个团队在开发真正有用的东西。

实习结束之后，他们给了我一个 offer，摆在我面前的有两个选择：一是我可以回学校继续读两年的书，毕业以后再去找我已有经验的工作，二是继续之前那份工作。对于我来说，我觉得自己已经有了开发背景，并且爱我现在的工作，学校也离得近，可以继续和以前的朋友保持联系，所以干脆辍学。这真的得分人，不能说“上大学没意义，不要上大学”之类的话，个人不是很赞同。

RW：很多人还不知道，其实在你创办 GitHub 之前已经做了 Gravatar 服务，背后的理念是什么?

TP-W：那个谈不上公司，只是我自己在做顾问期间所做的一个副产品。当时博客兴起，许多开发者和设计师都在写博客，我就想可以做点什么事情呢？那时候每天早上醒来都会花一个小时的时间盯着天花板想到底该怎么做。某天无意中想起 Web 论坛中每个人一般都有个头像，在评论中能把头像显示出来，但博客那时候还没有这种功能，所以就想，做一个吧。

做好之后就开始让朋友尝试，有些人确实很喜欢，但有一段时间其实并没有多少用户。最后终于受到许多人欢迎，尽管如此，这个产品也给了我很大压力，因为这个东西没有商业模式，我自己承担运营和基础设施建设的费用，要处理规模化中一些很棘手的问题。然后会因为稳定性不够系统瘫痪了而受到用户批评，从那以后学到了许多东西，主要就是如果你要做一个项目，一定要首先考虑好商业模式。

在把 Gravatar 卖给 Automattic 之后，我开始思考我的下一个副产品该做什么，商业模式应该是怎样的？最终怎么变现？如果做好了自己是不是能全职去做？也会评估项目的价值。

RW：2008 年，Git 其实已经很老了，你怎么会围绕 Git 开发一个协作社区？

TP-W：那时候在我负责运营的 Ruby 社区，Git 已经开始流行了，它具备 Subversion（另一个开源代码版本控制系统）所不具备的功能，并且这些功能正好是开源爱好者使用的，也是 Ruby 社区现在使用最多的功能。

Git 之前发展确实不温不火，命令行接口也相当复杂，但支持许多分支管理和分

布式协作，能让每个软件的克隆版本拥有完整的历史记录。当时就觉得这个东西将来人们一定会用到，他们没有理由不用这个，只是时间问题而已。

回想起来，使用 Git 最困难的部分是上传和分享 repo 文件，你必须有 Linux 服务器，然后要新建一个账号，下载代码生成 SSH-Key，反正非常痛苦。所以我和一些朋友就聚在一起说，我们要做一个简单的东西，方便人们分享 Git repo 文件。因为我们是 Web 开发者，所以我们就要建一个基于 Web 的东西，我们可以自己用，也要开源了让别人用，虽然现在 Git 不是太受欢迎，但 Linux 得用它，这是个好现象，如果我们让 Git 变得简单易用，其他人就想去用，最终 Git 就能受到欢迎。这就是当时的想法，趁人们还没意识到 Git 潜力的时候，先做一个产品去抢占这个市场，然后在这个市场上成为领跑者。

RW：GitHub 也做了很多工作去教初学者如何使用 Git，那么 GitHub 是如何增加用户量的?

TP-W：我们做了许多培训材料，有一个培训团队专门做这些，告诉人们如何使用 Git 和 GitHub，我们一直都在努力让网站变得更简单，不仅是为软件开发者，也为那些自己使用软件的人，但我们主要还是关注开发者。

为了简化 GitHub 的使用，我们做过的最重要的一件事情就是让大家通过 Web 接口去使用 Git。2008 年那会儿，使用的时候必须去下载命令行接口到本地设备中，然后通过命令行弄明白如何使用它，花了大量的精力，如果你不是一个软件开发者，你可能根本就不会碰这个东西。如果你对 Git 不熟悉，它真的很复杂。

最后我们为 Mac 和 Windows 平台开发了客户端，使它能被开发者、设计师、版本控制的新手，甚至硬核开发者等等喜欢使用图形界面的人所使用。但真正让 GitHub 被广为使用的，还是我们去年所做的事情，你可以通过网络添加文件、删除文件、修改文件，不用下载任何东西，这个功能很强大，让人们可以访问并编辑文件或者共同写博客。

RW：你觉得到目前为止，GitHub 最大的改变是什么？

TP-W：其实改变在很多方面，员工数量已经增加到了 217 人，然后我们已经两年没有办公室了，现在终于有了办公室。不过这些都是表面的。

RW：你曾说过要通过 GitHub 开源一切，怎么会有这种想法？这个事情的应该是 GitHub 和政府先做。你们下一步会怎么做？

TP-W：我们最初有许多想法，至于为什么想把政府所做的事情开放，就是因为觉得民众可以从开放的政府中获益很多。你想如果美国政府真的服务于民众，那么就应该尽量公开他们所做的事情。比如如何制定了法律、如何用法律与民众沟通。如果人们能看见法律在朝一个好的方向发展，他们参与度就越高。我们现在有工具、有互联网支持你去分享，你也可以让法律简单易懂。

让政府开放是其中一个想法，另外两个分别是科学和教育，所以我们会有专人去各大高校和中学讲 GitHub、软件开发、版本控制、行业变化、如何让东西变得更加精细易用等等。这对学生来说好处很多，如果能找到更好的方法把事情做得更优质更快，就可以腾出更多时间关心其他事情。

开放对于科学也有很大机会，在科研当中几乎没有人会写一个软件作为研究本身的一部分，那么其他人如果想利用研究中的一些东西就非常难。

原文 http://www.csdn.net/article/2013-11-28/2817647?utm_source=Tuicool_Weekly

Atlas 2.0.0 发布，来自 360 的 MySQL 中间层

Atlas 今天发布了 2.0 版本，改进内容包括：

1. support long connection
2. remove min-idle-connections
3. SQL log can be closed
4. remove the mode restriction of config file
5. fix space bug in admin.lua
6. remove chassis_private->cons and con_mutex
7. remove is_insert_id
8. mysql-proxyd reports error info when startup fails
9. check_state doesn't change the state of backends to down

下载地址：https://github.com/Qihoo360/Atlas/releases

Atlas 是由 Qihoo 360, Web 平台部基础架构团队开发维护的一个基于 MySQL 协议的数据中间层项目。它在 MySQL 官方推出的 MySQL-Proxy 0.8.2 版本的基础上，修改了大量 bug，添加了很多功能特性。目前该项目在 360 公

司内部得到了广泛应用，很多 MySQL 业务已经接入了 Atlas 平 台，每天承载的读写请求数达几十亿条。

Atlas 的详细介绍：请点这里

Atlas 的下载地址：请点这里

原文 http://www.oschina.net/p/atlas

百度开源平台上线，聚合百度开源项目

百度开源开源项目 EChartsUEditor

摘要：百度公司近日上线“百度开源平台”，该平台主要用来展示百度的开源项目。 目前该平台列出了 10 款百度公司的开源项目。

百度公司近日上线“百度开源平台”，该平台主要用来展示百度的开源项目。

目前该平台列出了 10 款百度公司的开源项目，分别有：

- Terminator：一款服务器虚拟化解决方案
- Itest：面向 service 接口的自动化测试工具，可用于集成测试或者系统级

测试

- UEditor：一款所见即所得富文本 Web 编辑器
- ECharts：基于 Canvas，纯 JavaScript 图表库，可用来个性化定制数据可视化图表
- ESUI：一套简单的 UI Library，提供一系列的控件，能满足基本页面交互功能
- F.I.S： 全称为 Front-end Integrated Solution，一套完整的前端技术解决方案，包括前段框架、模板框架、自动化框架以及辅助开发工具
- EDP：一个企业级前端应用的开发平台
- EST：一个基于 LESS 的样式工具库，帮助开发者更轻松地书写 LESS 代码
- ER：一个富浏览器端 web 应用的框架，可以方便地构建一个整站式的 AJAX web 应用
- Tangram：一款实用的 JavaScript 基础库，可以迅速构建出高度互动的 Web 应用程序
- Cafe：一款 Android 平台的自动化测试框架，框架覆盖了 Android 自动化测试的各种需求

之前百度开源的项目都托管在 Github 上，并且是以团队为主导，比较分散，如今推出的开源平台将这些项目集中展示，**这说明在公司层面，百度已经开始重视开源。**

如今国内企业在开源方面的投入越来越大，尤其是大型互联网企业。大企业的这些项目，基本上都已经在线上产品中投入使用，已经得到了充分的测试，以开源

形式发布后，其他开发者可以拿来就用，避免了重复制造轮子，且一般大企业产品的生命周期比较长，开发者不用担心这个项目会突然消失。

BAT 三大巨头参与开源（百度开源平台、阿里开源项目、腾讯开源项目），这对于国内开源生态也有很好的促进作用，可以让更多的人理解开源、参与开源。

原文 http://www.csdn.net/article/2013-11-29/2817656?utm_source=Tuicool_Weekly

Kraken 改变 PayPal 开发文化的 Node.js 框架

摘要：PayPal（全球流行的网上支付服务）公司近期发布了一款 Node.js Web 开发框架——Kraken，该框架基于 Express，并在此基础上提供了更加稳健的功能合集，支持本地化、环境配置、更加注重应用程序安全等。

PayPal（全球流行的网上支付服务）公司近期发布了一款 Node.js Web 开发框架——Kraken。

Kraken 基于 Express，Express 是目前 Node.js 上最流行的 MVC 模式的 Web 开发框架，通过提供一系列强大特性帮助开发者快速创建各种 Web 应用。而 Kraken 在 Express 的基础上提供了更加稳健的功能合集，支持本地化、环境配置、更加注重应用程序安全等。

为什么会有 Kraken?

之前，PayPal 公司长期存在着“非我所创”的文化，这导致 PayPal 采用新技术的态度很消极，项目开发进度也极其缓慢。正是由于 PayPal 行动缓慢，其他支付服务商 Stripe 和 Square 趁机成长，逐渐撼动 PayPal 的市场地位。同时，PayPal 当时的开发技术也已经无法满足快速开发的需求，因为当时的开发基本全是 Java，不需要用 Java 来实现的也会用 Java 完成。

2012 年 4 月，David Marcuss 成为 PayPal 的总裁，并任命工程师团队在 6 周内完成支付系统的重写，这是一个为 PayPal 带来了 35 亿美元收入的系统。最终，工程师团队用了 8 周时间完成了该项任务，他们选择了 Node.js 对系统进行重新开发。当然，PayPal 的其他大量的子系统还需要整合到 Node.js 系统，所以起初 Node.js 仅作为一个快速开发原型架构。

后来，PayPal 越来越多的新开发项目都逐渐采用 Node.js 和其他开源软件来完成，这就促成了一个可以快速开发 Web 应用的 Kraken.js 项目的诞生。

下面我们对 Kraken 的框架结构和特点进行简单的总结分析，希望能够帮助大家更好地了解 Kraken。

Kraken 框架套件

Kraken 框架套件包括多个部分，Kraken.js 仅是该框架的主体部分。该框架还包括其他模块（可独立使用）：

- lusca：支持 Express 的应用程序安全模块
- makara：支持 Dust.js 的国际化（i18n）模块
- Adaro：支持 Express 的一款 Dust.js 视图渲染器
- Kappa：NPM 代理插件

除了上述可以独立使用的模块之外，Kraken 套件还包括了一系列的依赖和实用工具：

- Generator-kraken：Yeoman 生成器
- Enrouten：用于 Express 的路由（route）配置中间件（初始化与配置模块）
- Kraken-devtools：Kraken 应用程序开发所需的工具合集

Kraken 的特点

通过前面的介绍，我们已经基本了解了 Kraken 究竟是怎么一回事儿，那 Kraken 到底具有哪些吸引开发者的特点呢?

1. 项目结构清晰

Kraken 将生成的项目的配置、内容和模板、路由逻辑（routing logic）放在了不同的位置，方便开发者对文件进行组织和管理。下面，我们详细了解下 Kraken 所创建项目的基本结构：

- /config，存放应用程序和中间件配置
- /controllers，控制器
- /lib，存放开发者自定义的库文件和其他代码
- /locales，特定语言内容
- /models，模型
- /public，公共的网络资源
- /public/templates，服务器和浏览器端模板
- /test，存放单元和功能测试用例等
- index.js，应用程序入口文件

随着项目的不断成长，这种组织方式和策略对开发者来说会更加友好。

2. 良好平衡开发环境与生产环境

Kraken 的配置文件为 /config/app.json，它会在运行时加载文件中包含的键值对。全部的应用程序配置均存放在这一个文件中，大大方便了开发者。/config/middleware.json 则是自定义的中间件配置文件。

说起开发环境与生产环境，二者通常在端口号、主机名等方面的参数设置会有不同，Kraken 允许创建开发模式下使用的配置文件，如 /config/app.json 为生产环境下的配置，/config/app-development.json 则是开发环境下的配置，

然后可以通过自定义环境变量（定义环境变量 NODE_ENV 为 production 或 development）来控制要加载哪个配置文件。如此一来，可以方便开发者在生产环境和开发环境进行快速切换！

3. 注重安全

通过 Lusca 模块来为安全保驾护航，并遵循 OWASP 安全原则，同时也默认为全部调用启用了几个请求/响应头文件：

- 跨站请求伪造（CSRF）标头
- 内容安全策略（CPS）标头
- 隐私优先项目平台（P3P）标头
- X-FRAME-OPTIONS 防止点击劫持

4. 其他

同时，它还有下面几个特点：

- 路由（route）与逻辑（logic）分离：方便程序运行失败时快速锁定问题。
- 模板（template）共用：Kraken 选用 Dust 作为模板语言，同时在服务器端和客户端使用了同一个模板，如此一来，就可以做到代码复用。
- 支持本地化： Kraken 可以在运行时基于请求上下文来加载内容包（content bundle），所以在模板渲染之前就可以通过添加上下文来给用户提供相应的本地语言支持，大大增强了应用的友好性。

示例与文档

关于 Kraken，官方给出了下面两个示例，可以帮助大家学习和了解：

例一：本地化与国际化：给 Web 应用增加多语言支持，让各地用户能够用当地语言来使用 Web 应用

例二：部署中间件：创建一个网页计数器，为大家解释在应用生命周期中如何部署中间件

开源项目

Kraken 是一款开源项目（分发协议为 Apache License v2.0），大家都可以对该项目进行 Fork 和提交修改。它也提供了一份简洁的使用文档。项目源码和文档均可在 Github 页面进行查看。

最后

任何一个项目的诞生都有其产生的背景，也必然有相应的需求驱动。正如 Kraken，正是由于 PayPal 为了满足其快速开发 Web 应用的强烈需求，才选择了 Node.js，才有了这个项目。

同时，我们也需要注意工程师团队中存在的“非我所创”文化，对于新技术要采取积极的态度，不能一成不变，否则将给企业发展带来阻碍。

原文

http://www.csdn.net/article/2013-11-25/2817617-PayPal-Kraken-Nodejs-Framework?utm_source=Tuicool_Weekly

JavaScript 核心

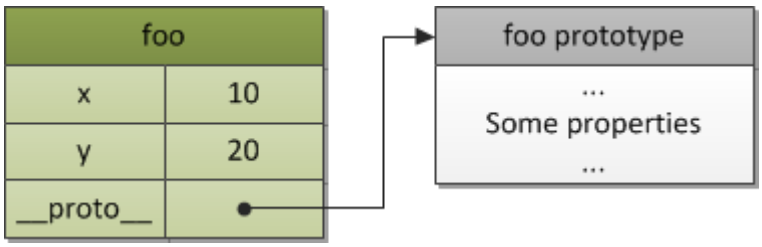
这篇文章是「深入 ECMA-262-3」系列的一个概览和摘要。每个部分都包含了对应章节的链接，所以你可以阅读它们以便对其有更深的理解。
面向读者：经验丰富的程序员，专家。
我们以思考对象的概念做为开始，这是 ECMAScript 的基础。

对象

ECMAScript 做为一个高度抽象的面向对象语言，是通过*对象*来交互的。即使 ECMAScript 里边也有*基本类型*，但是，当需要的时候，它们也会被转换成对象。一个对象就是一个属性集合，并拥有一个独立的 prototype（原型）对象。这个 prototype 可以是一个对象或者 null。
让我们看一个关于对象的基本例子。一个对象的 prototype 是以内部的 [[Prototype]]属性来引用的。但是，在示意图里边我们将会使用 __<internal-property>__下划线标记来替代两个括号，对于 prototype 对象来说是：__proto__。
对于以下代码：

```
var foo = {
x: 10,
y: 20
};
```

我们拥有一个这样的结构，两个明显的自身属性和一个隐含的__proto__属性，这个属性是对 foo 原型对象的引用：



这些 prototype 有什么用？让我们以*原型链*（prototype chain）的概念来回答这个问题。

原型链

原型对象也是简单的对象并且可以拥有它们自己的原型。如果一个原型对象的原型是一个非 null 的引用，那么以此类推，这就叫作*原型链*。
原型链是一个用来实现继承和共享属性的有限对象链。
考虑这么一个情况，我们拥有两个对象，它们之间只有一小部分不同，其他部分都相同。显然，对于一个设计良好的系统，我们将会*重用*相似的功能/代码，而不是在每个单独的对象中重复它。在基于类的系统中，这个代码*重用*风格叫作*类*

*继承*一你把相似的功能放入类 A 中，然后类 B 和类 C 继承类 A，并且拥有它们自己的一些小的额外变动。

ECMAScript 中没有类的概念。但是，代码重用的风格并没有太多不同（尽管从某些方面来说比基于类（class-based）的方式要更加灵活）并且通过*原型链*来实现。这种继承方式叫作*委托继承*(delegation based inheritance)（或者，更贴近 ECMAScript 一些，叫作*原型继承*(prototype based inheritance)）。跟例子中的类 A，B，C 相似，在 ECMAScript 中你创建对象：a，b，c。于是，对象 a 中存储对象 b 和 c 中通用的部分。然后 b 和 c 只存储它们自身的额外属性或者方法。

```
var a = {
x: 10,
calculate: function (z) {
return this.x + this.y + z
}
};

var b = {
y: 20,
__proto__: a
};

var c = {
y: 30,
__proto__: a
};

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80
```

足够简单，是不是？我们看到 b 和 c 访问到了在对象 a 中定义的 calculate 方法。这是通过原型链实现的。

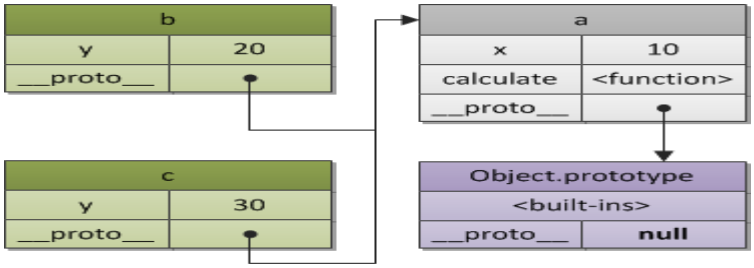
规则很简单：如果一个属性或者一个方法在对象*自身*中无法找到（也就是对象自身没有一个那样的属性），然后它会尝试在原型链中寻找这个属性/方法。如果这个属性在原型中没有查找到，那么将会查找这个原型的原型，以此类推，遍历整个原型链（当然这在类继承中也是一样的，当解析一个继承的*方法*的时候一我们遍历 *class 链*（ class chain））。第一个被查找到的同名属性/方法会被使用。因此，一个被查找到的属性叫作*继承*属性。如果在遍历了整个原型链之后还是没有查找到这个属性的话，返回 undefined 值。

注意，继承方法中所使用的 this 的值被设置为*原始*对象，而并不是在其中查找到这个方法的（原型）对象。也就是，在上面的例子中 this.y 取的是 b 和 c 中的值，

而不是 a 中的值。但是，this.x 是取的是 a 中的值，并且又一次通过*原型链*机制完成。

如果没有明确为一个对象指定原型，那么它将会使用__proto__的默认值－Object.prototype。Object.prototype 对象自身也有一个__proto__属性，这是原型链的*终点*并且值为 null。

下一张图展示了对象 a，b，c 之间的继承层级：



注意： ES5 标准化了一个实现原型继承的可选方法，即使用 Object.create 函数：

```
var b = Object.create(a, {y: {value: 20}});
var c = Object.create(a, {y: {value: 30}});
```

你可以在对应的章节获取到更多关于 ES5 新 API 的信息。 ES6 标准化了 __proto__属性，并且可以在对象初始化的时候使用它。

通常情况下需要对象拥有*相同或者相似的状态结构*（也就是相同的属性集合），赋以不同的*状态值*。在这个情况下我们可能需要使用*构造函数*(constructor function)，其以*指定的模式*来创造对象。

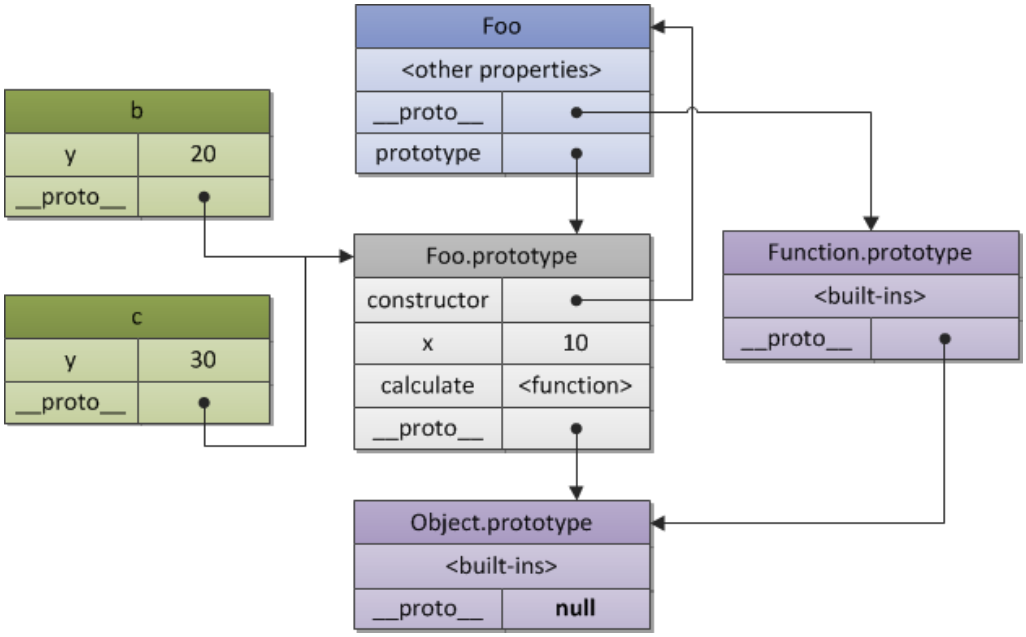
构造函数

除了以指定模式创建对象之外，*构造函数*也做了另一个有用的事情－它*自动地为新创建的对象设置一个原型对象*。这个原型对象存储在 ConstructorFunction.prototype 属性中。

换句话说，我们可以使用构造函数来重写上一个拥有对象 b 和对象 c 的例子。因此，对象 a（一个原型对象）的角色由 Foo.prototype 来扮演：

```
// a constructor function
function Foo(y) {
// which may create objects
// by specified pattern: they have after
// creation own "y" property
this.y = y;
}

// also "Foo.prototype" stores reference
// to the prototype of newly created objects,
// so we may use it to define shared/inherited
// properties or methods, so the same as in
// previous example we have:
```

```
// inherited property "x"
Foo.prototype.x = 10;

// and inherited method "calculate"
Foo.prototype.calculate = function (z) {
return this.x + this.y + z;
};

// now create our "b" and "c"
// objects using "pattern" Foo
var b = new Foo(20);
var c = new Foo(30);

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80

// let's show that we reference
// properties we expect

console.log(

b.__proto__ === Foo.prototype, // true
c.__proto__ === Foo.prototype, // true

// also "Foo.prototype" automatically creates
// a special property "constructor", which is a
// reference to the constructor function itself;
// instances "b" and "c" may found it via
// delegation and use to check their constructor

b.constructor === Foo, // true
c.constructor === Foo, // true
Foo.prototype.constructor === Foo // true

b.calculate === b.__proto__.calculate, // true
b.__proto__.calculate === Foo.prototype.calculate // true

);
```
这个代码可以表示为如下关系：



这张图又一次说明了每个对象都有一个原型。构造函数 Foo 也有自己的__proto__，值为 Function.prototype，Function.prototype 也通过其__proto__属性关联到 Object.prototype。因此，重申一下，Foo.prototype 就是 Foo 的一个明确的属性，指向对象 b 和对象 c 的原型。

正式来说，如果思考一下*分类*的概念（并且我们已经对 Foo 进行了*分类*），那么构造函数和原型对象合在一起可以叫作「类」。实际上，举个例子，Python 的*第一级*（first-class）动态类（dynamic classes）显然是以同样的属性/方法处理方案来实现的。从这个角度来说，Python 中的类就是 ECMAScript 使用的委托继承的一个语法糖。

注意：在 ES6 中「类」的概念被标准化了，并且实际上以一种构建在构造函数上面的语法糖来实现，就像上面描述的一样。从这个角度来看原型链成为了类继承的一种具体实现方式：

```
// ES6
class Foo {
constructor(name) {
this._name = name;
}

getName() {
return this._name;
}
}

class Bar extends Foo {
getName() {
return super.getName() + ' Doe';
```

```
}
}

var bar = new Bar('John');
console.log(bar.getName()); // John Doe
```

有关这个主题的完整、详细的解释可以在 ES3 系列的第七章找到。分为两个部分：7.1 面向对象.基本理论，在那里你将会找到对各种面向对象范例、风格的描述以及它们和 ECMAScript 之间的对比，然后在 7.2 面向对象.ECMAScript 实现，是对 ECMAScript 中面向对象的介绍。

现在，在我们知道了对象的基础之后，让我们看看*运行时程序的执行*（runtime program execution）在 ECMAScript 中是如何实现的。这叫作*执行上下文栈*（execution context stack），其中的每个元素也可以抽象成为一个对象。是的，ECMAScript 几乎在任何地方都和对象的概念打交道;)

执行上下文堆栈

这里有三种类型的 ECMAScript 代码：*全局*代码、*函数*代码和 *eval* 代码。每个代码是在其*执行上下文*（execution context）中被求值的。这里只有一个全局上下文，可能有多个函数执行上下文以及 *eval* 执行上下文。对一个函数的每次调用，会进入到函数执行上下文中，并对函数代码类型进行求值。每次对 eval 函数进行调用，会进入 *eval* 执行上下文并对其代码进行求值。

注意，一个函数可能会创建无数的上下文，因为对函数的每次调用（即使这个函数递归的调用自己）都会生成一个具有新状态的上下文：

```
function foo(bar) {}

// call the same function,
// generate three different
// contexts in each call, with
// different context state (e.g. value
// of the "bar" argument)

foo(10);
foo(20);
foo(30);
```

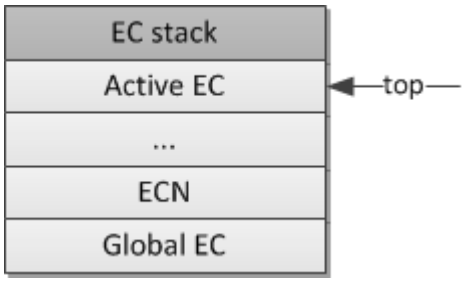
一个执行上下文可能会触发另一个上下文，比如，一个函数调用另一个函数（或者在全局上下文中调用一个全局函数），等等。从逻辑上来说，这是以栈的形式实现的，它叫作*执行上下文栈*。

一个触发其他上下文的上下文叫作 *caller*。被触发的上下文叫作 *callee*。callee 在同一时间可能是一些其他 callee 的 caller（比如，一个在全局上下文中被调用的函数，之后调用了一些内部函数）。

当一个 caller 触发（调用）了一个 callee，这个 caller 会暂缓自身的执行，然

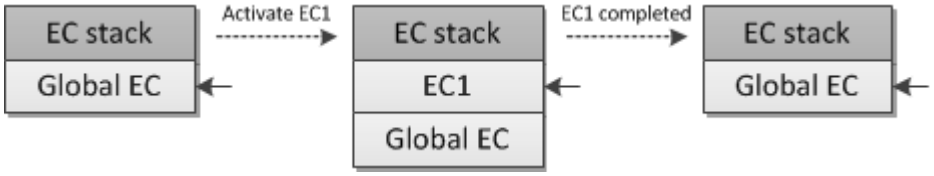
后把控制权传递给 callee。这个 callee 被 push 到栈中，并成为一个*运行中*（活动的）执行上下文。在 callee 的上下文结束后，它会把控制权返回给 caller，然后 caller 的上下文继续执行（它可能触发其他上下文）直到它结束，以此类推。callee 可能简单的*返回*或者由于*异常*而退出。一个抛出的但是没有被捕获的异常可能退出（从栈中 pop）一个或者多个上下文。

换句话说，所有 ECMAScript *程序的运行时*可以用*执行上下文（EC）栈*来表示，*栈顶*是当前*活跃*(active)上下文：



当程序开始的时候它会进入*全局执行上下文*，此上下文位于*栈底*并且是栈中的*第一个*元素。然后全局代码进行一些初始化，创建需要的对象和函数。在全局上下文的执行过程中，它的代码可能触发其他（已经创建完成的）函数，这些函数将会进入它们自己的执行上下文，向栈中 push 新的元素，以此类推。当初始化完成之后，运行时系统（runtime system）就会等待一些*事件*（比如，用户鼠标点击），这些事件将会触发一些函数，从而进入新的执行上下文中。

在下个图中，拥有一些函数上下文 EC1 和全局上下文 Global EC，当 EC1 进入和退出全局上下文的时候下面的栈将会发生变化：



这就是 ECMAScript 的运行时系统如何真正地管理代码执行的。

更多有关 ECMAScript 中执行上下文的信息可以在对应的第一章 执行上下文中获取。

像我们所说的，栈中的每个执行上下文都可以用一个对象来表示。让我们来看看它的结构以及一个上下文到底需要什么*状态*（什么属性）来执行它的代码。

执行上下文

一个执行上下文可以抽象的表示为一个简单的对象。每一个执行上下文拥有一些属性（可以叫作*上下文状态*）用来跟踪和它相关的代码的执行过程。在下图中展示了一个上下文的结构：

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[Variable object + all parent scopes]
thisValue	Context object

除了这三个必需的属性（一个*变量对象*（variable objec），一个 *this* 值以及一个*作用域链*（scope chain））之外，执行上下文可以拥有任何附加的状态，这取决于实现。

让我们详细看看上下文中的这些重要的属性。

变量对象

变量对象是与执行上下文相关的数据作用域。它是一个与上下文相关的特殊对象，其中存储了在上下文中定义的变量和函数声明。

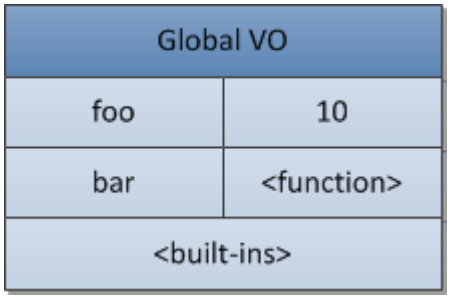
注意，*函数表达式*（与*函数声明*相对）*不包含*在变量对象之中。

变量对象是一个抽象概念。对于不同的上下文类型，在物理上，是使用不同的对象。比如，在全局上下文中变量对象就是*全局对象本身*（这就是为什么我们可以通过全局对象的属性名来关联全局变量）。

让我们在全局执行上下文中考虑下面这个例子：

```
var foo = 10;

function bar() {} // function declaration, FD
(function baz() {}); // function expression, FE

console.log(
this.foo == foo, // true
window.bar == bar // true
);

console.log(baz); // ReferenceError, "baz" is not defined
```

之后，全局上下文的变量对象（variable objec，简称 VO）将会拥有如下属性：



再看一遍，函数 baz 是一个*函数表达式*，没有被包含在变量对象之中。这就是为什么当我们想要在函数自身之外访问它的时候会出现 ReferenceError。

注意，与其他语言（比如 C/C++）相比，在 ECMAScript 中*只有函数*可以创建一个新的作用域。在函数作用域中所定义的变量和内部函数在函数外边是不能直接访问到的，而且并不会污染全局变量对象。

使用 eval 我们也会进入一个新的（eval 类型）执行上下文。无论如何，eval 使用全局的变量对象或者使用 caller（比如 eval 被调用时所在的函数）的变量对象。

那么函数和它的变量对象是怎么样的？在函数上下文中，变量对象是以*活动对象*（activation object）来表示的。

活动对象

当一个函数被 caller *所触发*（被调用），一个特殊的对象，叫作*活动对象*（activation object）将会被创建。这个对象中包含*形参*和那个特殊的 arguments 对象（是对形参的一个映射，但是值是通过索引来获取）。*活动对象*之后会做为函数上下文的*变量对象*来使用。

换句话说，函数的变量对象也是一个同样简单的变量对象，但是除了变量和函数声明之外，它还存储了形参和 arguments 对象，并叫作*活动对象*。

考虑如下例子：

```
function foo(x, y) {
var z = 30;
function bar() {} // FD
(function baz() {}); // FE
}

foo(10, 20);
```

我们看下函数 foo 的上下文中的活动对象（activation object，简称 AO）：

Activation object	
x	10
y	20
arguments	{0: 10, 1: 20, ..}
z	30
bar	<function>

并且*函数表达式* baz 还是没有被包含在变量/活动对象中。

关于这个主题所有细节方面（像变量和函数声明的*提升问题*（hoisting））的完整描述可以在同名的章节第二章 变量对象中找到。

注意，在 ES5 中*变量对象*和*活动对象*被并入了*词法环境*模型（lexical environments model），详细的描述可以在对应的章节找到。

然后我们向下一个部分前进。众所周知，在 ECMAScript 中我们可以使用*内部函数*，然后在这些内部函数我们可以引用*父*函数的变量或者*全局*上下文中的变量。当我们把变量对象命名为上下文的*作用域对象*，与上面讨论的原型链相似，这里有一个叫作*作用域链*的东西。

作用域链

作用域链是一个对象列表，上下文代码中出现的标识符在这个列表中进行查找。这个规则还是与原型链同样简单以及相似：如果一个变量在函数自身的作用域（在自身的变量/活动对象）中没有找到，那么将会查找它父函数（外层函数）的变量对象，以此类推。

就上下文而言，标识符指的是：变量*名称*，函数声明，形参，等等。当一个函数在其代码中引用一个不是局部变量（或者局部函数或者一个形参）的标识符，那么这个标识符就叫*作自由变量*。*搜索这些自由变量*(free variables)正好就要用到*作用域链*。

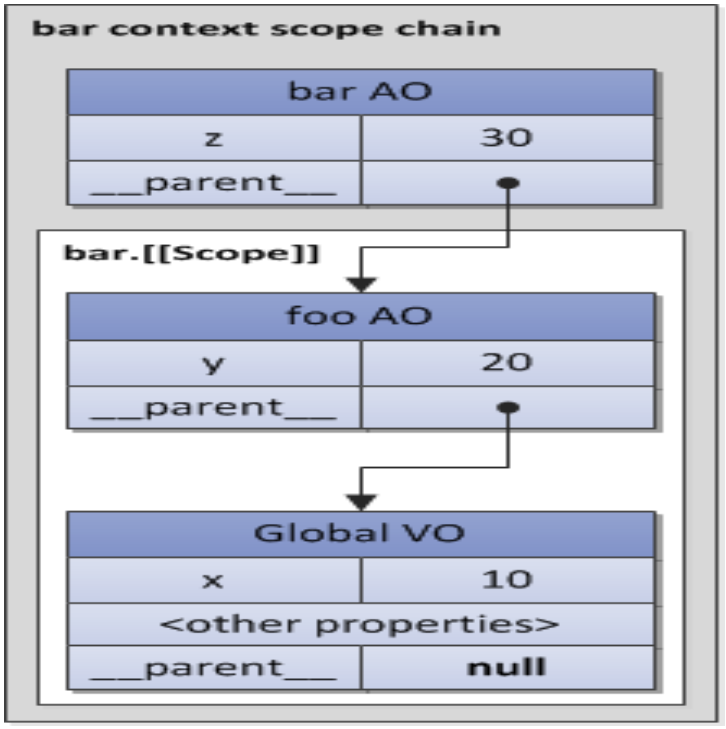
在通常情况下，*作用域链*是一个包含所有*父（函数）变量对象*__*加上*（在作用域链头部的）函数*自身变量/活动对象*的一个列表。但是，这个作用域链也可以包含任何其他对象，比如，在上下文执行过程中动态加入到作用域链中的对象－像 *with 对象*或者特殊的 *catch 从句*（catch-clauses）对象。

当*解析*（查找）一个标识符的时候，会从作用域链中的活动对象开始查找，然后（如果这个标识符在函数自身的活动对象中没有被查找到）向作用域链的上一层查找－重复这个过程，就和原型链一样。

```
var x = 10;

(function foo() {
```

```
var y = 20;
(function bar() {
var z = 30;
// "x" and "y" are "free variables"
// and are found in the next (after
// bar's activation object) object
// of the bar's scope chain
console.log(x + y + z);
})();
})();
```

我们可以假设通过隐式的__parent__属性来和作用域链对象进行关联，这个属性指向作用域链中的下一个对象。这个方案可能在真实的 Rhino 代码中经过了测试，并且这个技术很明确得被用于 ES5 的词法环境中（在那里被叫作 outer 连接）。作用域链的另一个表现方式可以是一个简单的数组。利用__parent__概念，我们可以用下面的图来表现上面的例子（并且父变量对象存储在函数的[[Scope]]属性中）：



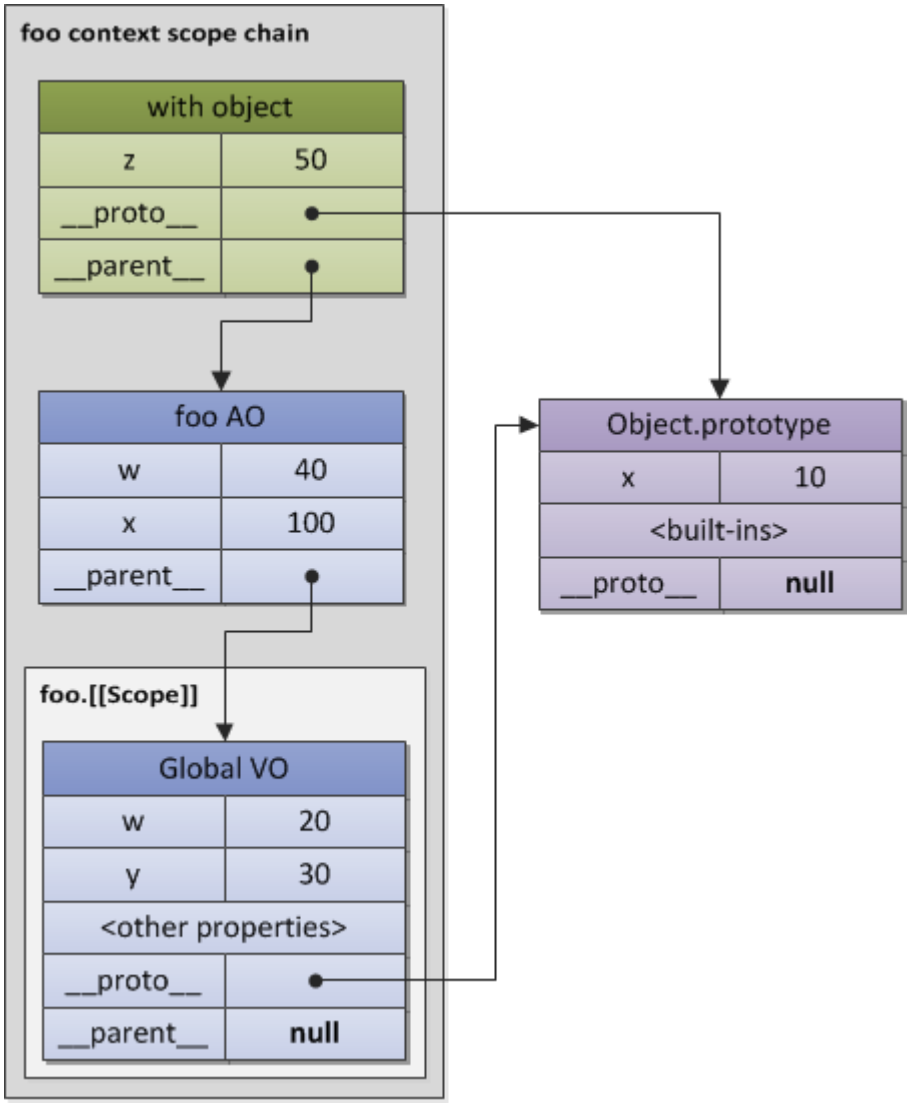
在代码执行过程中，作用域链可以通过使用 with 语句和 catch 从句对象来增强。并且由于这些对象是简单的对象，它们可以拥有原型（和原型链）。这个事实导致作用域链查找变为*两个维度*：（1）首先是作用域链连接，然后（2）在每个作用域链连接上－深入作用域链连接的原型链（如果此连接拥有原型）。

对于这个例子：

```
Object.prototype.x = 10;

var w = 20;
```

```
var y = 30;

// in SpiderMonkey global object
// i.e. variable object of the global
// context inherits from "Object.prototype",
// so we may refer "not defined global
// variable x", which is found in
// the prototype chain

console.log(x); // 10

(function foo() {

// "foo" local variables
var w = 40;
var x = 100;

// "x" is found in the
// "Object.prototype", because
// {z: 50} inherits from it

with ({z: 50}) {
console.log(w, x, y , z); // 40, 10, 30, 50
}

// after "with" object is removed
// from the scope chain, "x" is
// again found in the AO of "foo" context;
// variable "w" is also local
console.log(x, w); // 100, 40

// and that's how we may refer
// shadowed global "w" variable in
// the browser host environment
console.log(window.w); // 20

})();
```

我们可以给出如下的结构（确切的说，在我们查找__parent__连接之前，首先查找__proto__链）：



注意，不是在所有的实现中全局对象都是继承自 Object.prototype。上图中描述的行为（从全局上下文中引用「未定义」的变量 x）可以在诸如 SpiderMonkey 引擎中进行测试。

由于所有父变量对象都存在，所以在内部函数中获取父函数中的数据没有什么特别－我们就是遍历作用域链去解析（搜寻）需要的变量。就像我们上边提及的，在一个上下文结束之后，它所有的状态和它自身都会被*销毁*。在同一时间父函数可能会*返回*一个*内部函数*。而且，这个返回的函数之后可能在另一个上下文中被调用。如果自由变量的上下文已经「消失」了，那么这样的调用将会发生什么？通常来说，有一个概念可以帮助我们解决这个问题，叫作*（词法）闭包*，其在 ECMAScript 中就是和*作用域链*的概念紧密相关的。

闭包

在 ECMAScript 中，函数是*第一级*（first-class）对象。这个术语意味着函数可以做为参数传递给其他函数（在那种情况下，这些参数叫作「函数类型参数」（funargs，是"functional arguments"的简称））。接收「函数类型参数」

的函数叫作*高阶函数*或者，贴近数学一些，叫作高阶*操作符*。同样函数也可以从其他函数中返回。返回其他函数的函数叫作*以函数为值*（function valued）的函数（或者叫作拥有*函数类值*的函数（functions with functional value））。这有两个在概念上与「函数类型参数（funargs）」和「函数类型值（functional values）」相关的问题。并且这两个子问题在*"Funarg problem"*（或者叫作"functional argument"问题）中很普遍。为了解决*整个"funarg problem"*，*闭包*（closure）的概念被创造了出来。我们详细的描述一下这两个子问题（我们将会看到这两个问题在 ECMAScript 中都是使用图中所提到的函数的[[Scope]]属性来解决的）。

「funarg 问题」的第一个子问题是*「向上 funarg 问题」*（upward funarg problem）。它会在当一个函数从另一个函数向上返回（到外层）并且使用上面所提到的*自由变量*的时候出现。为了在*即使父函数上下文结束*的情况下也能访问其中的变量，内部函数在*被创建的时候*会在它的[[Scope]]属性中保存父函数的*作用域链*。所以当函数被*调用*的时候，它上下文的作用域链会被格式化成活动对象与[[Scope]]属性的和（实际上就是我们刚刚在上图中所看到的）：

Scope chain = Activation object + [[Scope]]

再次注意这个关键点－确切的说在*创建时刻*－函数会保存*父函数的*作用域链，因为确切的说这个*保存下来的作用域链*将会在未来的函数调用时用来查找变量。

```
function foo() {
var x = 10;
return function bar() {
console.log(x);
};
}

// "foo" returns also a function
// and this returned function uses
// free variable "x"

var returnedFunction = foo();

// global variable "x"
var x = 20;

// execution of the returned function

returnedFunction(); // 10, but not 20
```

这个类型的作用域叫作*静态（或者词法）作用域*。我们看到变量 x 在返回的 bar 函数的[[Scope]]属性中被找到。通常来说，也存在*动态作用域*，那么上面例子中的变量 x 将会被解析成 20，而不是 10。但是，动态作用域在 ECMAScript 中没

有被使用。
「funarg 问题」的第二个部分是*「向下 funarg 问题」*。这种情况下可能会存在一个父上下文，但是在解析标识符的时候可能会模糊不清。问题是：标识符该使用*哪个作用域*的值－以静态的方式存储在函数创建时刻的还是在执行过程中以动态方式生成的（比如 *caller* 的作用域）？为了避免这种模棱两可的情况并形成闭包，*静态作用域*被采用：

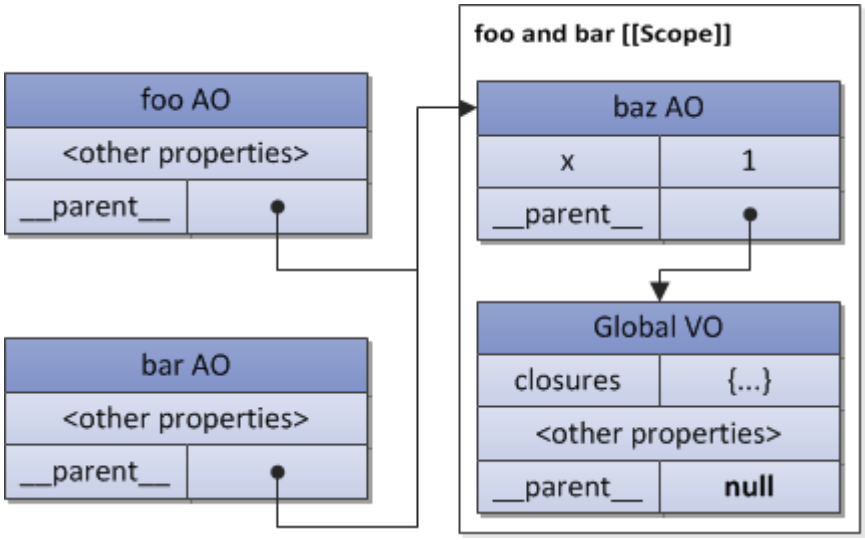
```
// global "x"
var x = 10;

// global function
function foo() {
console.log(x);
}

(function (funArg) {

// local "x"
var x = 20;

// there is no ambiguity,
// because we use global "x",
// which was statically saved in
// [[Scope]] of the "foo" function,
// but not the "x" of the caller's scope,
// which activates the "funArg"

funArg(); // 10, but not 20

})(foo); // pass "down" foo as a "funarg"
```

我们可以断定*静态作用域*是一门语言拥有*闭包的必需条件*。但是，一些语言可能会同时提供动态和静态作用域，允许程序员做选择－什么应该包含（closure）在内和什么不应包含在内。由于在 ECMAScript 中只使用了静态作用域（比如我们对于 funarg 问题的两个子问题都有解决方案），所以结论是：*ECMAScript 完全支持闭包*，技术上是通过函数的[[Scope]]属性实现的。现在我们可以给闭包下一个准确的定义：
闭包是一个代码块（在 ECMAScript 是一个函数）和以静态方式/词法方式进行存储的所有父作用域的一个集合体。所以，通过这些存储的作用域，函数可以很容易的找到自由变量。
注意，由于*每个*（标准的）函数都在创建的时候保存了[[Scope]]，所以理论上来讲，ECMAScript 中的*所有函数*都是*闭包*。
另一个需要注意的重要事情是，多个函数可能拥有*相同的父作用域*（这是很常见

的情况，比如当我们拥有两个内部/全局函数的时候）。在这种情况下，[[Scope]]属性中存储的变量是在拥有相同父作用域链的*所有函数之间共享*的。一个闭包对变量进行的修改会*体现*在另一个闭包对这些变量的读取上：

```
function baz() {
var x = 1;
return {
foo: function foo() { return ++x; },
bar: function bar() { return --x; }
};
}

var closures = baz();

console.log(
closures.foo(), // 2
closures.bar() // 1
);
```

以上代码可以通过下图进行说明：



确切来说这个特性在循环中创建多个函数的时候会使人非常困惑。在创建的函数中使用循环计数器的时候，一些程序员经常会得到非预期的结果，所有函数中的计数器都是*同样*的值。现在是到了该揭开谜底的时候了－因为所有这些函数拥有同一个[[Scope]]，这个属性中的循环计数器的值是最后一次所赋的值。

```
var data = [];

for (var k = 0; k < 3; k++) {
data[k] = function () {
alert(k);
};
}
```

```
data[0](); // 3, but not 0
data[1](); // 3, but not 1
data[2](); // 3, but not 2
```

这里有几种技术可以解决这个问题。其中一种是在作用域链中提供一个额外的对象－比如，使用额外函数：

```
var data = [];

for (var k = 0; k < 3; k++) {
data[k] = (function (x) {
return function () {
alert(x);
};
})(k); // pass "k" value
}

// now it is correct
data[0](); // 0
data[1](); // 1
data[2](); // 2
```

对闭包理论和它们的实际应用感兴趣的同学可以在第六章 闭包中找到额外的信息。如果想获取更多关于作用域链的信息，可以看一下同名的第四章 作用域链。

然后我们移动到下个部分，考虑一下执行上下文的最后一个属性。这就是关于 this 值的概念。

This

this 是一个与执行上下文相关的特殊对象。因此，它可以叫作上下文对象（也就是用来指明执行上下文是在哪个上下文中被触发的对象）。

*任何对象*都可以做为上下文中的 this 的值。我想再一次澄清，在一些对 ECMAScript 执行上下文和部分 this 的描述中的所产生误解。this 经常被*错误的*描述成是变量对象的一个属性。这类错误存在于比如像这本书中（即使如此，这本书的相关章节还是十分不错的）。再重复一次：

this 是执行上下文的一个属性，而不是变量对象的一个属性。

这个特性非常重要，因为*与变量相反，this 从不会参与到标识符解析过程*。换句话说，在代码中当访问 this 的时候，它的值是*直接*从执行上下文中获取的，并*不需要任何作用域链查找*。this 的值只在*进入上下文*的时候进行*一次*确定。

顺便说一下，与 ECMAScript 相反，比如，Python 的方法都会拥有一个被当作简单变量的 self 参数，这个变量的值在各个方法中是相同的的并且在执行过程中可以被更改成其他值。在 ECMAScript 中，给 this 赋一个新值是*不可能的*，因为，再重复一遍，它不是一个变量并且不存在于变量对象中。

在全局上下文中，this 就等于*全局对象本身*（这意味着，这里的 this 等于*变量对*

象）：

```
var x = 10;

console.log(
x, // 10
this.x, // 10
window.x // 10
);
```

在函数上下文的情况下，对*函数的每次调用*，其中的 this 值可能是*不同的*。这个 this 值是通过*函数调用表达式*（也就是函数被调用的方式）的形式由 *caller* 所提供的。举个例子，下面的函数 foo 是一个 *callee*，在全局上下文中被调用，此上下文为 caller。让我们通过例子看一下，对于一个代码相同的函数，this 值是如何在不同的调用中（函数触发的不同方式），由 caller 给出*不同的*结果的：

```
// the code of the "foo" function
// never changes, but the "this" value
// differs in every activation

function foo() {
alert(this);
}

// caller activates "foo" (callee) and
// provides "this" for the callee

foo(); // global object
foo.prototype.constructor(); // foo.prototype

var bar = {
baz: foo
};

bar.baz(); // bar

(bar.baz)(); // also bar
(bar.baz = bar.baz)(); // but here is global object
(bar.baz, bar.baz)(); // also global object
(false || bar.baz)(); // also global object

var otherFoo = bar.baz;
otherFoo(); // again global object
```

为了深入理解 this 为什么（并且更本质一些－*如何*）在每个函数调用中可能会发生变化，你可以阅读第三章 This。在那里，上面所提到的情况都会有详细的讨

论。

总结

通过本文我们完成了对概要的综述。尽管，它看起来并不像是「概要」;)。对所有这些主题进行完全的解释需要一本完整的书。我们只是没有涉及到两个大的主题：*函数*（和不同函数之间的区别，比如，*函数声明*和*函数表达式*）和 ECMAScript 中所使用的*求值策略*(evaluation strategy)。这两个主题是可以 ES3 系列的在对应章节找到：第五章 函数和第八章 求值策略。

如果你有留言，问题或者补充，我将会很乐意地在评论中讨论它们。

祝学习 ECMAScript 好运！

原文

http://www.linuxeden.com/html/news/20131125/145873.html?utm_source=Tuicool_Weekly

想学响应式设计？来看史上最全的响应式设计资源库

响应式设计起源：Ethan Marcotte 在 2010 年写了一篇响应式的文章，宣扬

这种新式的网页设计思想，经过 3 年的发展，响应式设计得到了众多设计师的认可。

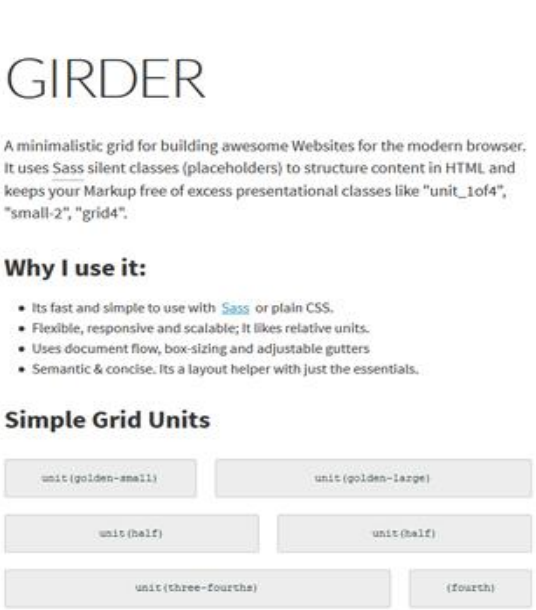
本文的目的在于为大家集中推荐一些最有价值的响应式设计资源。包含了 CSS 框架、在线工具、准备阶段的工具等等。

CSS 响应式框架

这一部分主要介绍了一些最新的 CSS 响应式框架，更轻量，而且兼容性也不差。（Foundation，Skeleton 是较为老式的 CSS 响应式框架）

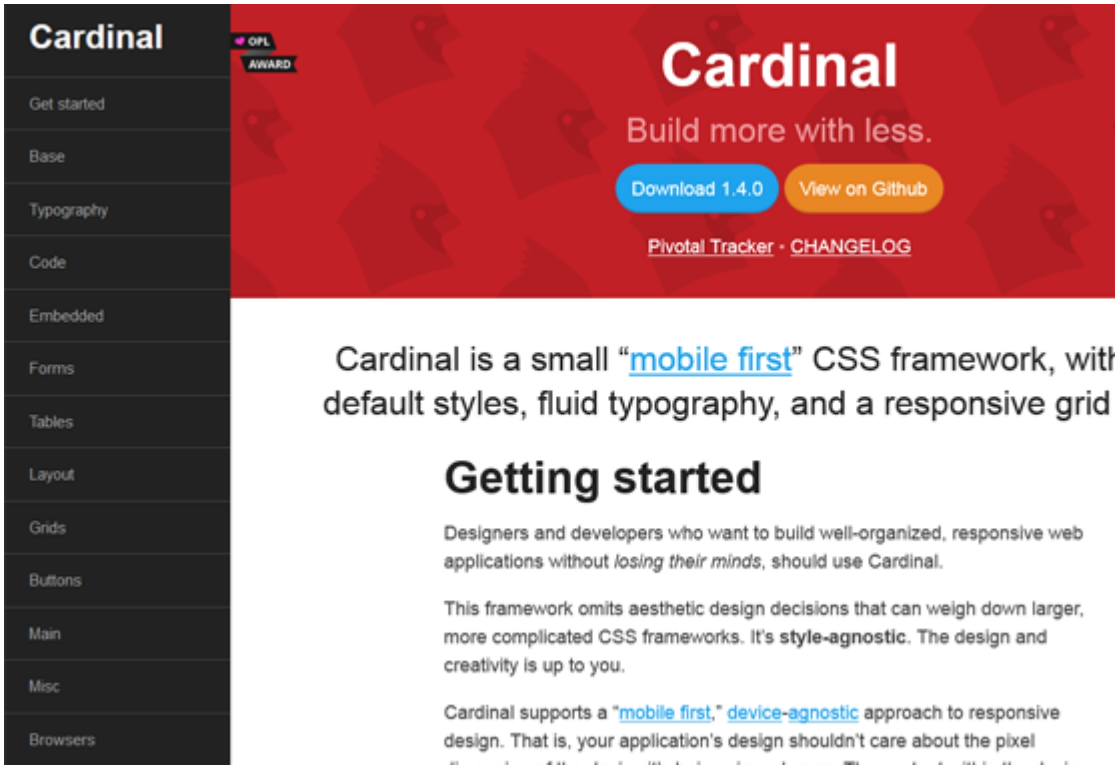
Girder

Girder 使用了 Sass silent classes (占位符，输出时不会体现)在 HTML 中组织内容，标记能够额外处理一些表象类（presentational classes ），比如“unit_1of4”，“small-2”，“grid4”.



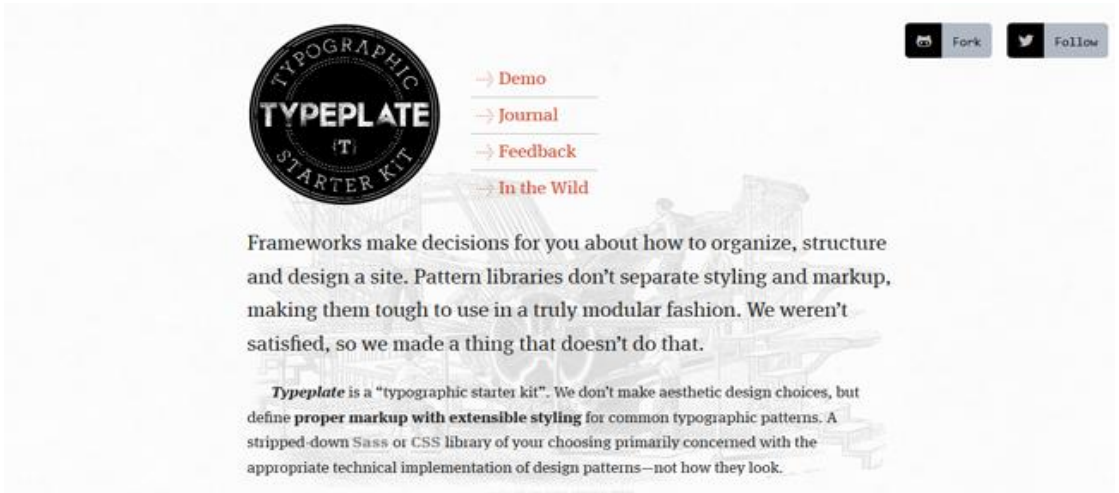
Cardinal

Cardinal 是一个小型的、以”移动优先”为理念的 CSS 框架，默认样式很实用。灵活的字体和响应式的栅格线系统。



Typeplate

Typeplate 是一款”排版初学者工具”．一款简约的 Sass/CSS 库，能够采用合适的手段处理作品。

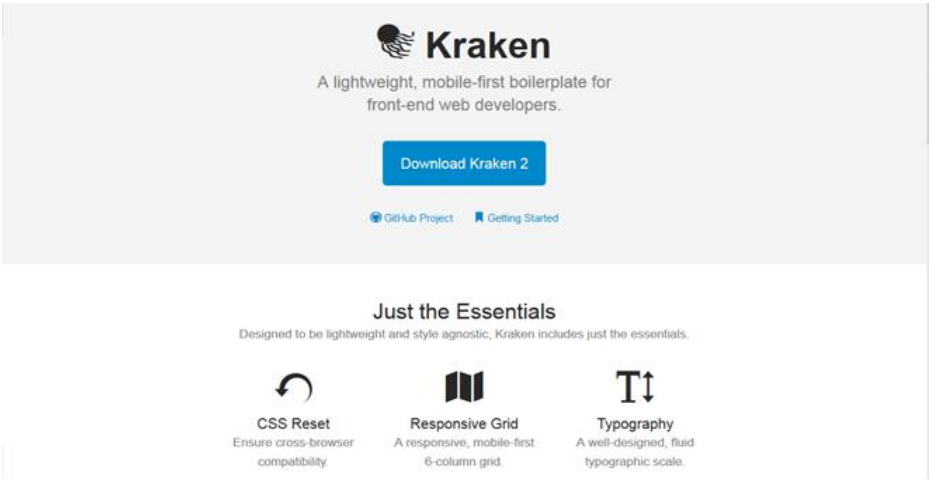


Furatto

Furatto 基于 Sass，包含了 JS 插件，在 Coffeescript 中开发的，因此很容易阅读。交互性很强，支持多种设备。

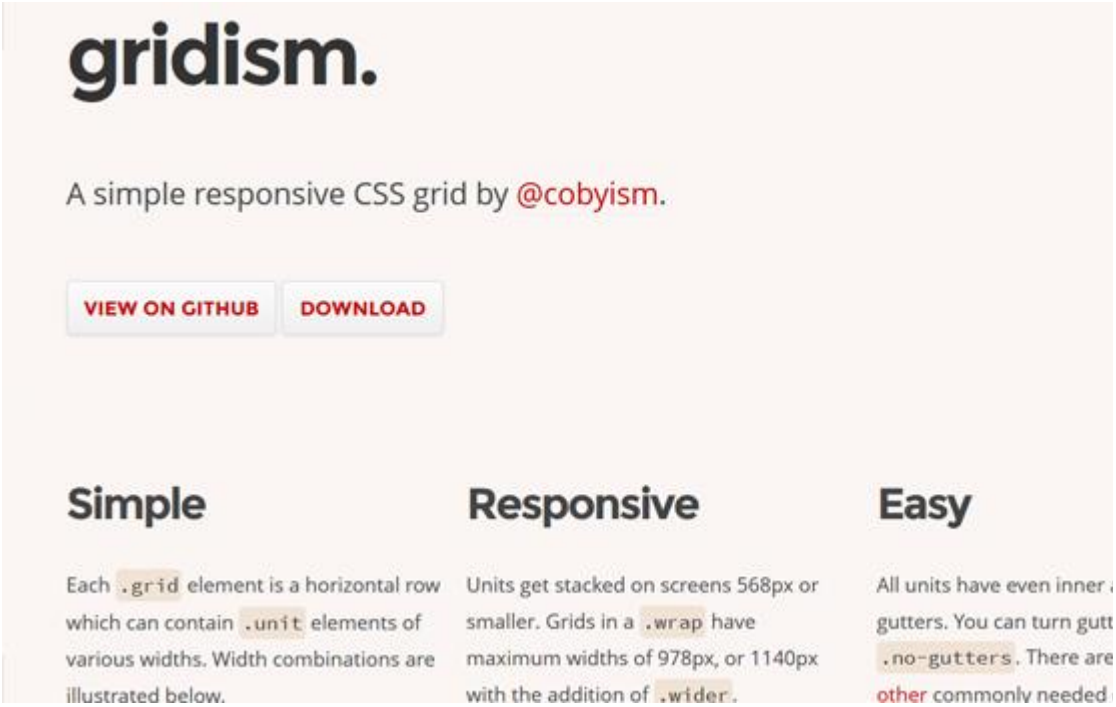
Kraken

轻量、移动优先为理念的模板，适合前端开发者。



Gridism

Gridism 是一款简约的响应式栅格线系统，非常好用。



Sassaparilla

Sassaparilla 使用了 Sass 以及 Compass 技术，让响应式网页设计变得简单。注重于更好的版式布局，打造良好的阅读节奏。让编译更加轻松。

Cool Kitten

Cool Kitten 是一款滚动视差响应式框架（个人最爱）

Responsive Boilerplate

Responsive Boilerplate 极度简约、非常轻量(2kb)的 CSS 栅格框架。易懂易用。

Javascript 响应式插件

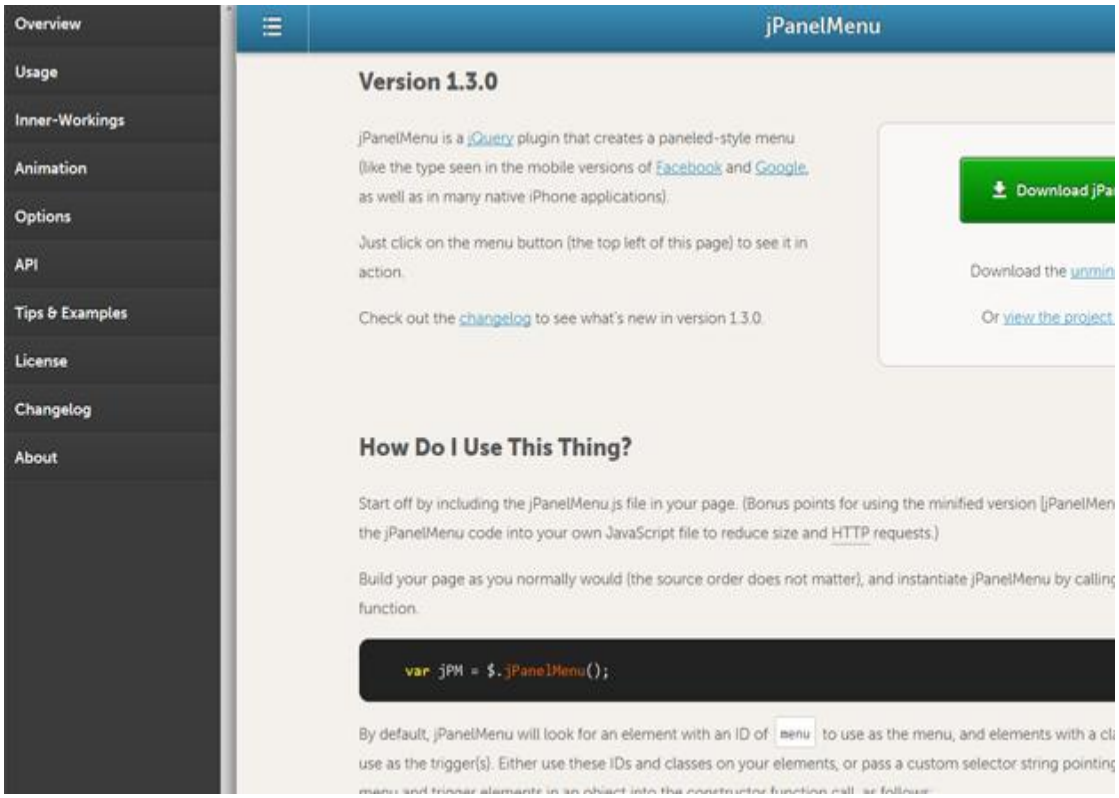
这些脚本要么是一小段代码，要么是几个文件，能够实现网页设计作品的响应式。大部分自动注释，因此很方便使用。

Responsive elements

Responsive Elements 小型 JS 库，能够轻松实现元素的响应式。

jPanelMenu

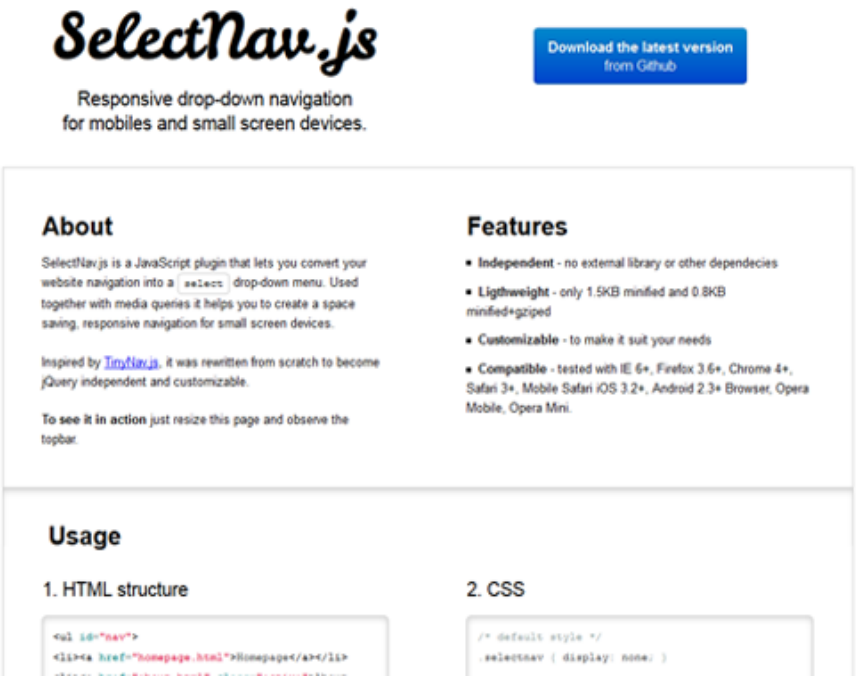
jPanelMenu 是一款 jQuery 插件，由一款面板式的菜单创建。保证 CSS 动画更好的实现。



SelectNav.js

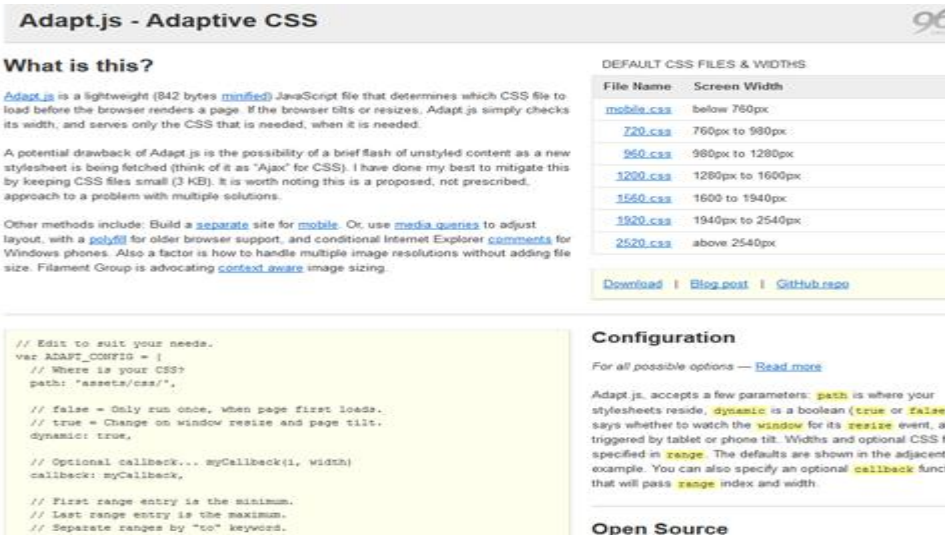
SelectNav 是一款 JS 插件，能够将网站的导航栏转换为可选择的下拉菜单。

结合了 Media Queries，能为移动设备节约空间。



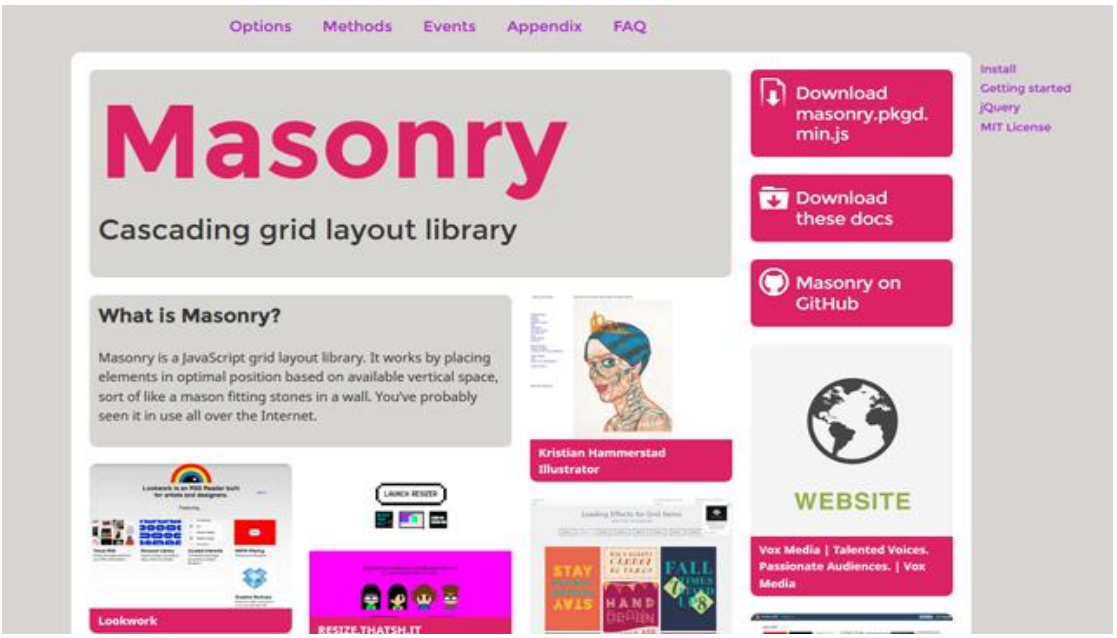
Adapt.js

Adapt.js 轻量级 JS 文件，在浏览器载入页面前，能够根据宽度，智能判定并载入 CSS 文件。



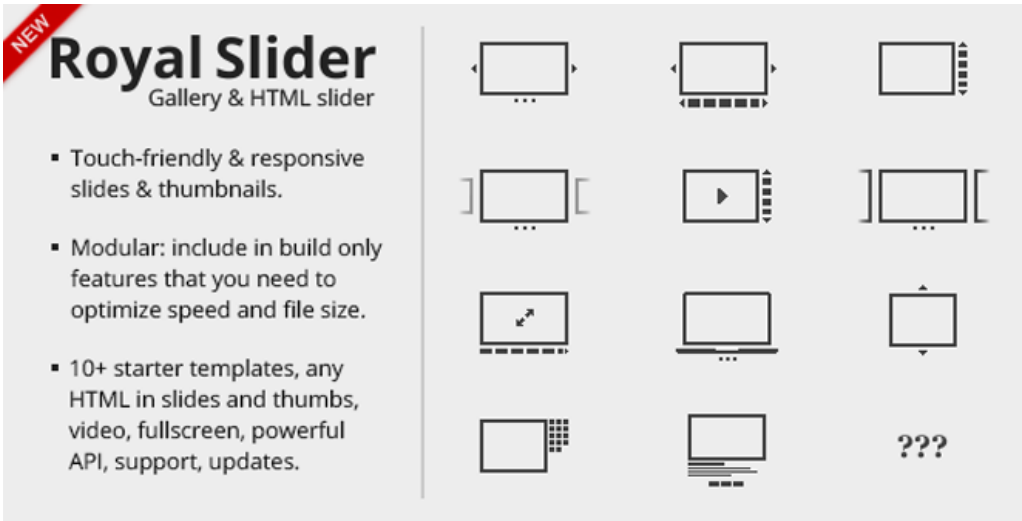
Masonry

Masonry 是一款优秀的 jQuery 插件，能够打造动态、适应性的布局。能够帮助重新排列元素。



RoyalSlider ($12)

RoyalSlider 是一款很好用的 jQuery 图库或内容滚动插件，动效、响应式布局、支持触控，很适合移动端。无论是滚动栏、幻灯片窗还是内容滚动栏、图库、视频库都能用其实现。



UberMenu – WordPress ($16)

UberMenu 是一款非常友好的、高度定制化的、响应式的 Mega Menu WordPress 插件。

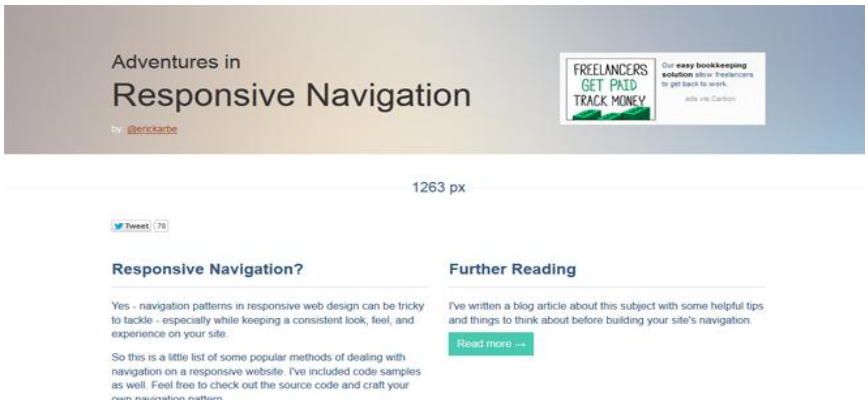
Responsive Pricing Tables – WordPress ($15)

CSS3 Responsive Web Pricing Tables Grids For WordPress CSS3 价格表，两种价格表样式，20 种可选颜色，可选项丰富。

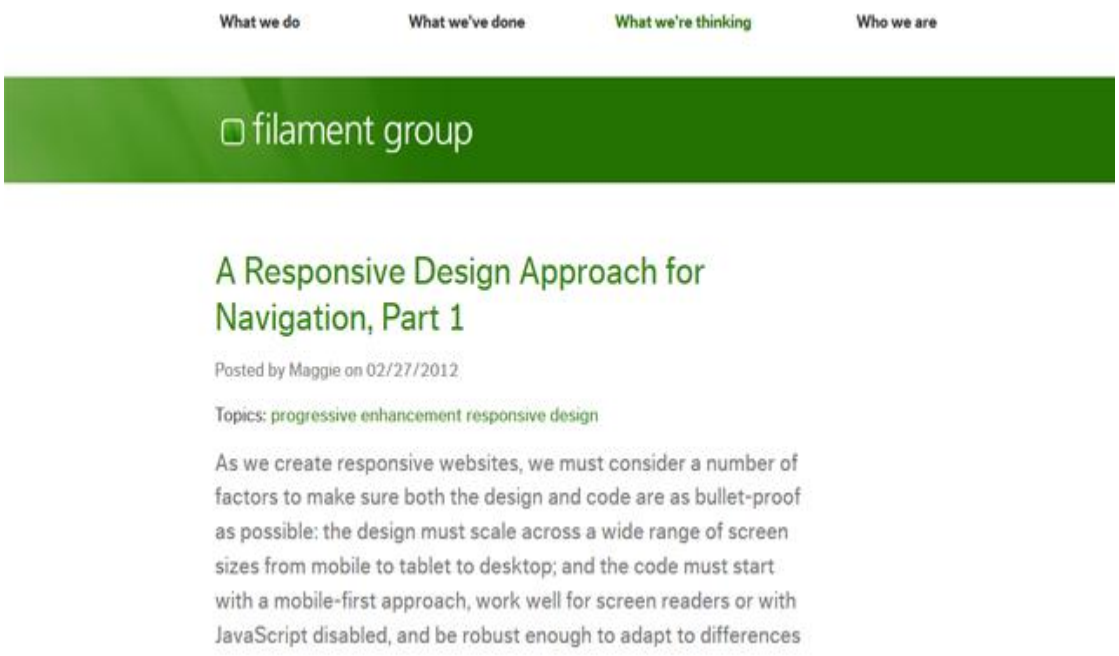
响应式设计——导航篇

响应式设计中最难的部分大概就是导航了。即便你的导航菜单占地很小，但是移动端的设计依然很困难。传统的网页设计总是把导航菜单放在页顶，以便用户浏览。而响应式设计中，情况很复杂。这一环节，将推荐几种不错的解决方案。

Responsive menu patterns



A Responsive Design Approach for Navigation (文章)

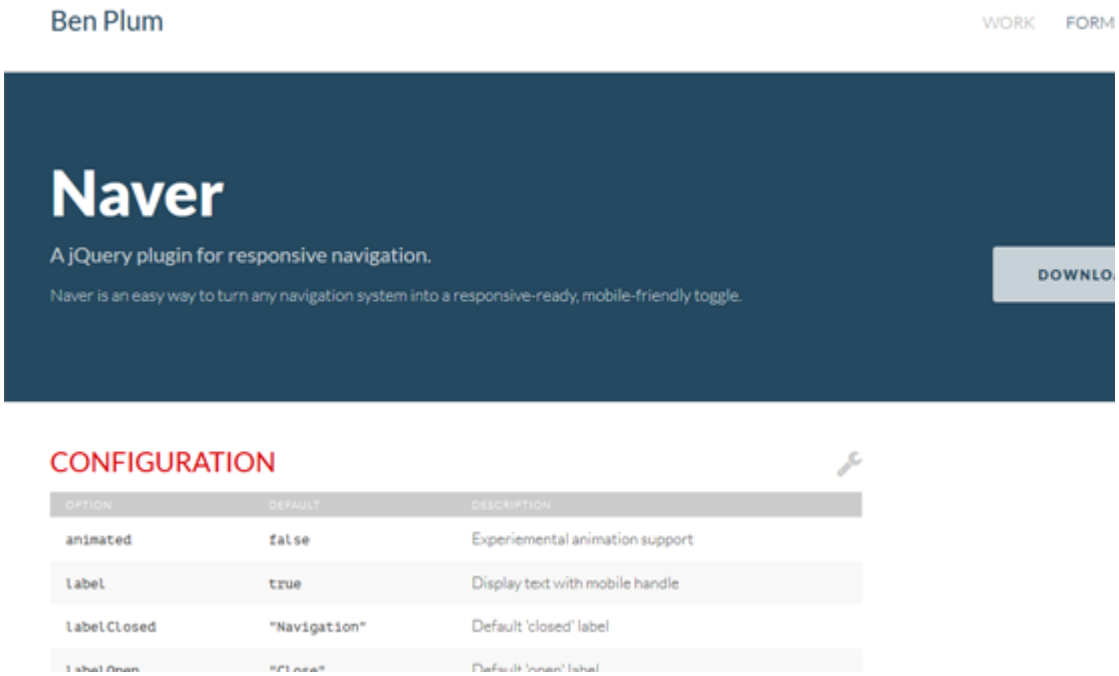


Flexnav

FlexNav 是一款使用 Media Queries 和 JS 实现的多级菜单，支持触控、悬停。符合”移动优先”理念。

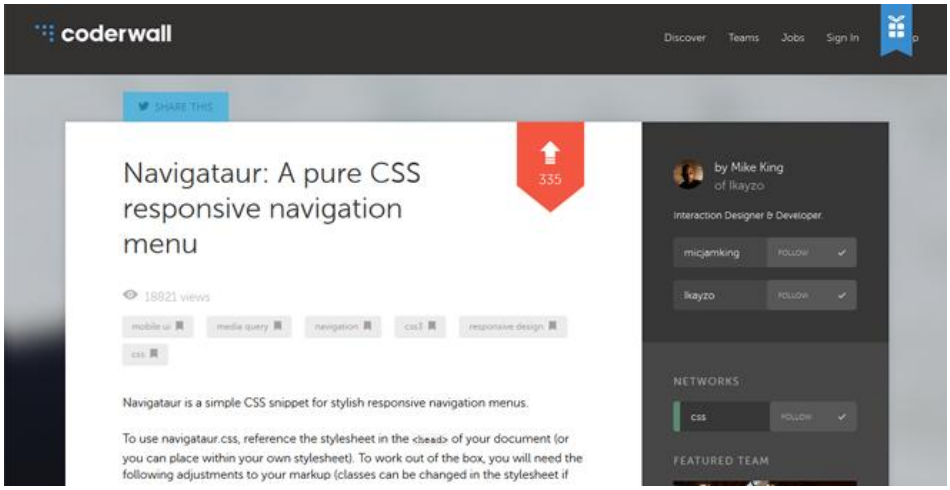
Naver

Naver 一款面向响应式导航的 jQuery 插件



Navigataur

Navigataur 是一款简约的工具，帮助实现响应式导航菜单。



响应式设计——图像

如何根据浏览器尺寸、加载速度选择图片尺寸?

CSS Fluid Image Techniques for Responsive Site Design (Article)

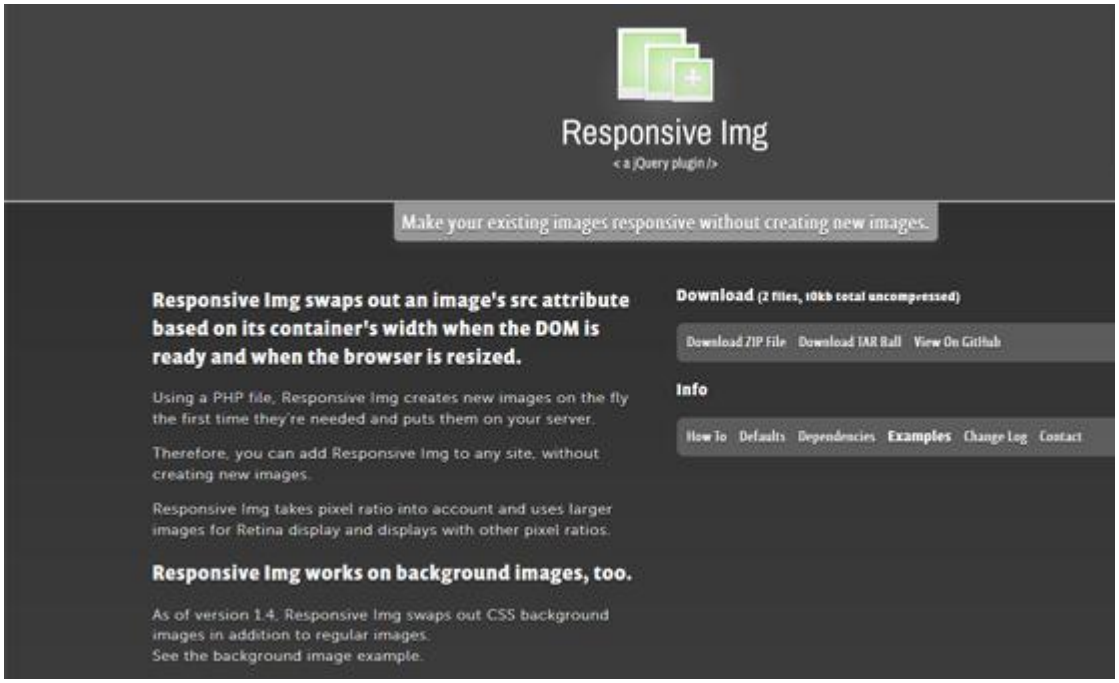
Dudley Storey 创作的一篇优秀文章，阐述了该要如何打造”伸缩自如”的图像

Clown Car Technique for Responsive Images

Estelle Weyl，写了一篇关于可缩放矢量图形(SVG)的文章，生动有趣。

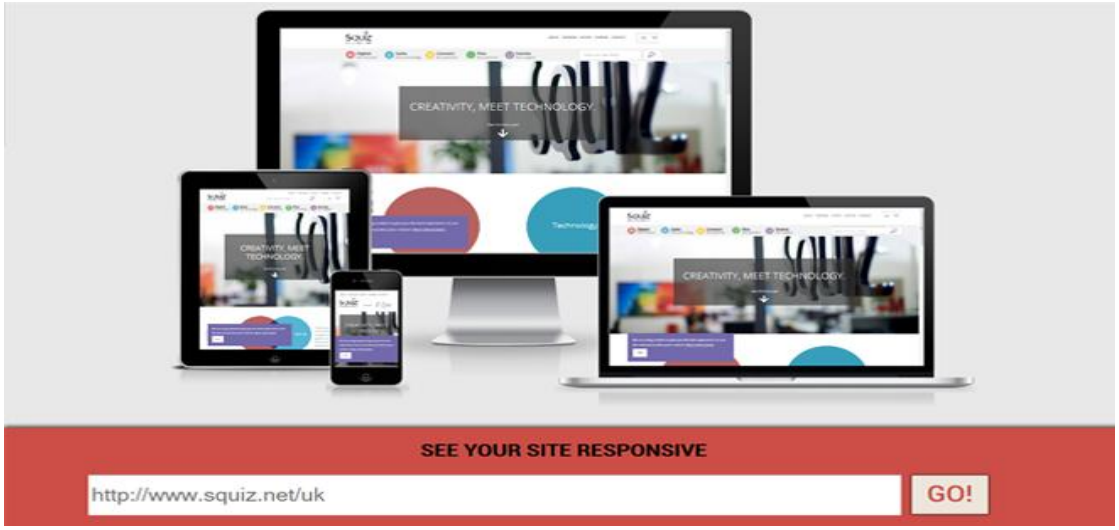
Responsive Img

Responsive Img 是一款 jQuery 插件，能够变换图像的 src 属性，主要根据的是容器的宽度。



响应式设计工具

See your site responsive

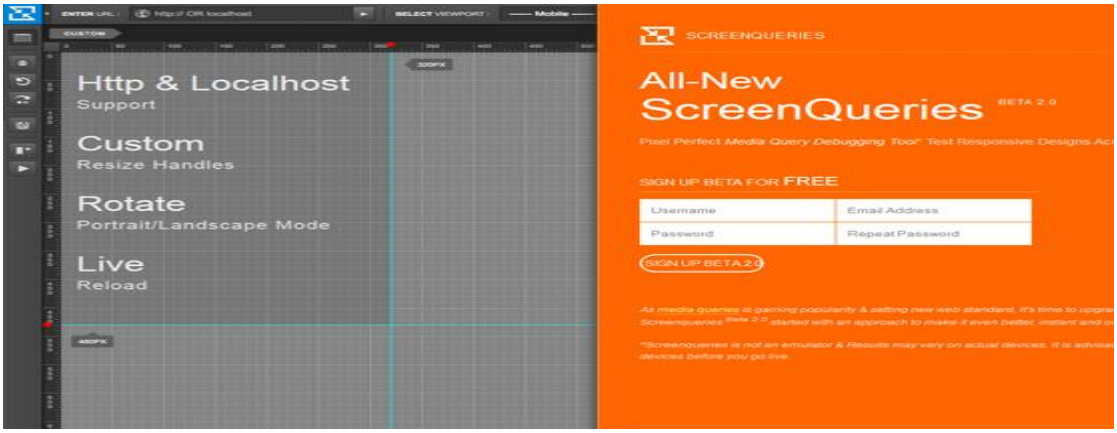


Viewport Resizer

Viewport Resizer 是一款在线工具能够测试网页设计是否符合响应式设计的标准。

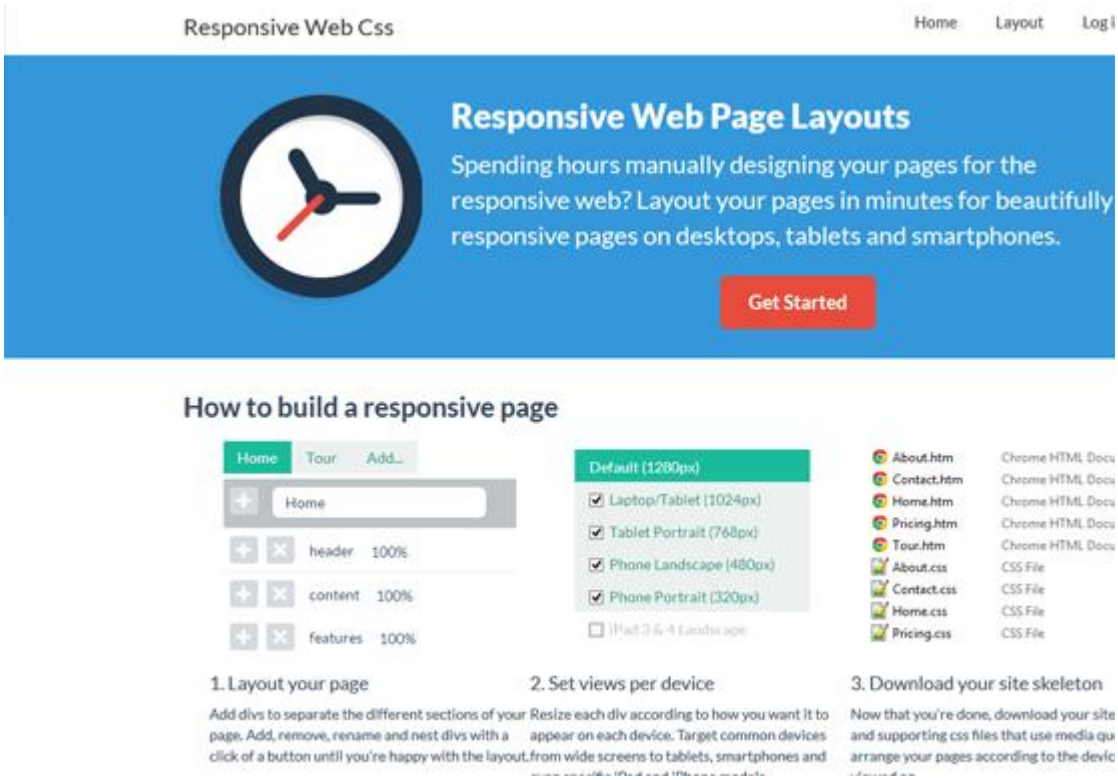
Screenqueri.es

Screenqueri.es 是一款细腻的响应式测试工具。能够在 30 多种不同的设备上展示你的网页设计。



Responsive Web CSS

Responsive Web CSS 让响应式网页布局变得轻松。只需要添加 div，设置一下不同设备上的大小即可。



原文

http://www.uisdc.com/ultimate-resources-to-responsive-design?utm_source=Tuicool_Weekly

前端性能优化的 14 个规则

作为一个半前端工程师，而且只会写点 HTML5 和 CSS3 的“假”前端工程师，为了能更好地理解一下前端的花花世界，最近拜读了《高性能网站建设指南》一

书，对作者提出的前端性能优化的 14 个规则获益匪浅，为了让自己印象更深刻点，决定作此文，当做学习笔记也好，知识总结也罢，总归看过的东西要让自己很好地掌握很好地运用起来才是王道。在解读这些规则的同时，我会用我一年半多的移动网站开发经历提出一些针对移动网站的优化建议。

规则 01：尽量减少 HTTP 请求

前端优化的黄金准则指导着前端页面的优化策略：只有 10%-20%的最终用户响应时间花在接受请求的 HTML 文档上，剩下的 80%-90%时间花在为 HTML 文档所引用的所有组件（图片、脚本、样式表等）进行的 HTTP 请求上。因此，改善响应时间的最简单途径就是减少组件的数量，并由此减少 HTTP 请求的数量。当然很多人就会说，既然这样，那我们就减少页面组件的数量不就 OK 了吗？那你试试，你会掀起一场性能优化和产品设计之间的大 PK。所以，我们要减少 HTTP 请求是要平衡性能和设计的。如果找到这个平衡点呢？书中从以下几个方面做了介绍，我逐一说明：

① 图片地图

初看“图片地图”四个字，对非专业的前端人员来说一头雾水，我的第一印象就是这样的，咱们以京东的移动站点为例，右侧用户和购物车的图标，正常实现我会选择如下方式：

```
<a href=”用户跳转页面 URL”>
        <div class=”定义用户 icon 显示的样式表”></div>
</a>
```

```
<a href=”购物车跳转页面 URL”>
        <div class=” 定义用户 icon 显示的样式表”></div>
</a>
```

这种方式无可厚非的，但是两张图片就有两个 HTTP 请求，这明显是增加了页面中的 HTTP 请求。

那么我们可以把这两个 HTTP 请求变成一个吗？答案当然是可以的，这就是图片地图：允许在一张图片上关联多个 URL，而目标 URL 的选择取决于用户单击了图片上的哪个位置。这样上面京东两个图标合并成一张图片，这样图片的 HTTP 请求就减少了一个。

示例代码如下：

```
<img src=”合并后的图片”>
<map name=”map1”>
        <area shape=”rect” coords=”0,0,40,40” href=”用户跳转页面 URL”>
        <area shape=”rect” coords=”50,0,90,40” href=”购物车跳转页面 URL”>
</map>
```

不过图片地图只支持矩形形状，其他形状不支持。

② 请 CSS 喝“雪碧”（CSS Sprites）

CSS Sprites 一句话：将多个图片合并到一张单独的图片，这样就大大减少了页面中图片的 HTTP 请求。

③ 内联图片和脚本

使用 data:URL（Base64 编码）模式直接将图片包含在 Web 页面中而无需进行 HTTP 请求。

但是此种方法存在明显缺陷：

- 不受 IE 的欢迎；

- 图片太大不宜采用这种方式，因为 Base64 编码之后会增加图片大小，这样页面整体的下载量会变大；

- 内联图片在页面跳转的时候不会被缓存。（大图片可以使用浏览器的本地缓存，在首次访问的时候保存到浏览器缓存中，典型的是 HTML5 的 manifest 缓存机制以及 LocalStorage 等）。

④ 样式表的合并

将页面样式定义、脚本、页面本身代码严格区分开，但是样式表、脚本也不是分割越细越好，因为没多引用一个样式表就增加一次 HTPP 请求，能合并的样式表尽量合并。一个网站有一个公用样式表定义，每个页面只要有一个样式表就 OK

啦。

通过以上四个努力之后，你会发现你的网页响应时间最多能减少一半，这不是作者说大话，也不是我狂吹，我亲手用我的移动网站首页做了一个尝试，本地测试之后响应时间能减少 40%左右。所以减少页面 HTTP 请求数量，是一个很重要的原则。遵循此原则可以同时改善首次访问和后续访问的响应时间，而每一个网站的首次响应时间会决定用户之后还来不来的重要原因。

规则 02：使用内容发布网络（CDN 的使用）

什么叫内容发布网络（CDN）？它是一组分布在多个不同地理位置的 Web 服务器，用于更加有效地向用户发布内容。主要用于发布页面静态资源：图片、css 文件、js 文件等。如此，能轻易地提高响应速度。

关于 CDN 的具体详细原理以及优缺点，各位可以自行询问度娘或者 google。

规则 03：添加 Expires 头

浏览器使用缓存来减少 HTTP 请求的数据，并减小 HTTP 响应的大小，使页面加载更快。Web 服务器使用 Expires 头来告诉浏览器它可以使用一个组件的当前副本，直到指定的 deadline 为止。HTTP 规范中称此头为：在这一时间之后响应被认为失效。

个人对这块表示不想使用，其实就是一句话，把一些 css、js、图片在首次访问

的时候全部缓存到浏览器本地，从我做移动网站的过程中发现，其实没有这么复杂，完全可以使用 HTML5 提供的本地缓存机制就 OK 了。关于 HTML5 本地缓存机制，各位可以查阅相关资料。后续我也会对 HTML5 的缓存机制进行介绍的。

规则 04：压缩组件（使用 Gzip 方式）

书中关于压缩从 gzip 压缩方式到如何压缩讲了很多，我想直接跳过，对于做 PC 网站或者移动网站来说，急需要压缩的是 css 文件和 js 文件，至于如何压缩，网上有很多在线工具，去挑选一个自己用的顺手看的顺眼的就好，当然也有人选择对 HTML 进行压缩，这样也可以。但是实际工作中我没有这么做。之所谓没有这么做，是因为我觉得很麻烦。不要鄙视我，毕竟我不是一个真正意义上的前端工程师，哈哈！

规则 05：将 CSS 样式表放在顶部

如果将 css 样式定义放在页面中或者页面底部，会出现短暂白屏或者某一区域短暂白板的情况，这和浏览器的运营机制有关的，不管页面如何加载，页面都是逐步呈现的。所以在每做一个页面的时候，用 Link 标签把每一个样式表定义放在 head 中。

规则 06：将 javascript 脚本放在底部

浏览器在加载 css 文件时，页面逐步呈现会被阻止，直到所有 css 文件加载完毕，

所以要把 css 文件的引用放到 head 中去，这样在加载 css 文件时不会组织页面的呈现。但是对于 js 文件，在使用的时候，它下面所有也页面内容的呈现都会被阻塞，将脚本放在页面越靠下的地方，就意味着越多的内容能够逐步呈现。

规则 07：避免使用 CSS 表达式

CSS 表达式是动态玩 CSS 的一种很强大的方式，但是强大的同时也存在很高的危险性。因为 css 表达式的频繁求值会导致 css 表达式性能低下。如果真想玩 css 表达式，可以选用只求值一次的表达式或者使用事件处理来改变 css 的值。

规则 08：使用外部 javascript 和 CSS

内联 js 和 css 其实比外部文件有更快的响应速度，那为什么还要用外部呢？因为使用外部的 js 和 css 可以让浏览器缓存他们，这样不仅 HTML 文档大小减少，而且不会增加 HTTP 请求数量。

另外，使用外部 js 和 css 可以提高组件的可复用性。

规则 09：减少 DNS 查询

DNS 查询有时间开销，通常一个浏览器查找一个给定主机名的 IP 地址需要 20-120ms。

缓存 DNS：缓存 DNS 查询可以很好地提高网页性能，一旦缓存了 DNS 查询，

之后对于相同主机名的请求就无需进行再次的 DNS 查找，至少短时间内不需要。

所以在使用页面中 URL、图片、js 文件、css 文件等时，不要使用过多不同的主机名。

规则 10：精简 javascript

如何精简？最初始的精简方式就是移除不必要的字符减小 js 文件大小，改善加载时间。包括所有的注释、不必要的空白字符。

高级一点的精简方式就是：混淆。它不但会移除不必要的字符，还会改写代码，比如函数和变量的名字会被改成很短的字符串，这样使 js 代码更简练更难阅读。

但是我一般很少使用混淆，一个现在互联网时代，代码没有必要整的那么神秘，大可以大家一起 share，天下代码一起抄，只要抄出自己的特色就 ok 了。而且一旦使用混淆，对于 js 代码的维护和调试都很复杂，因为有时候混淆之后的 js 代码完全看不懂。

其实实际开发过程中，从文件大小和代码可复用性来说，不仅仅是 js 代码需要精简，css 代码一样也很需要精简。

规则 11：避免重定向

重定向的英文是 Redirect，用于将用户从一个 URL 重新跳转到另一个 URL。最常见的 Redirect 就是 301 和 302 两种。

关于重定向的性能影响这里就不说了，自行查阅相关资料吧。

在我们实际开发中避免重定向最简单也最容易被忽视的一个问题就是，设置 URL 的时候，最后的“/”，有些人有时候会忽略，其实你少了“/”，这时候的 URL 就被重定向了，所以在给页面链接加 URL 的时候切记最后的“/”不可丢。

规则 12：删除重复脚本

重复的 js 代码除了有不必要的 HTTP 请求之外，还会浪费执行 js 的时间。

将你使用的 js 代码模块化，可以很好地避免这个问题，至于 js 模块化如何实现，现在有很多可以使用的开源框架，我用的比较多的是我们公司玉伯的 Sea.js。

规则 13：配置 ETag

Etag（Entity Tag），实体标签，是 Web 服务器和浏览器用户确认缓存组件的有效性的一种机制。

写的很复杂，对我这种非专业的前端开发人员来说，有点过了，关于这个原则有兴趣的自己看吧。

规则 14：使 Ajax 可缓存

针对页面中主动的 Ajax 请求返回的数据要缓存到本地，当然这个是针对短期内不会变化的数据。如果不确定数据变化周期的话，可以增加一个修改标识的判断，

我正常处理过程中会给一些 Ajax 请求返回的数据增加一个 MD5 值的判断，每次请求会判断当前 MD5 是否变化，如果变化了取最新的数据，如果不变化，则不变。

噼里啪啦说了一堆，14 个规则啊，那我们开发过程中要针对这每一个规则对自己的前端代码做 check 吗？我反正不这么干，做前端页面，尤其是移动网站的页面，我所记住的准则就是：尽量减少页面大小，尽量降低页面响应时间。在页面性能和交互设计之中找平衡点。

原文

http://www.cnblogs.com/iamjiuye/p/3446519.html?utm_source=Tuicool_Weekly

一个 Bump Pointer Allocator

最近抽时间在读 Milo Yip 翻译的《游戏引擎架构》。书还没有出版，我应邀给这个译本写序，所以先拿到了电子版，正在加紧读一遍。全书接近 800 页，我已经读了 600 页左右，希望这个周末可以抽时间读完。

这本书是由顽皮狗的主程之一 Jason Gregory 写的，内容很精彩，Milo Yip 翻译的也相当不错。有很多章节我读的很有共鸣，想先挑一点来写写。

由于作者的主要技术背景是 Console game 开发，而 Console 目前内存非常有限，且没有虚拟内存，对内存的管理和使用就非常苛刻。很多 PC 平台上几乎被忽视的问题到了 Console 平台上就需要仔细考虑了。我很有共鸣是因为

10 多年前在开发大话西游时，要求在 64M 内存上跑起来，同样写了好多内存管理相关的代码。

比如栈式内存管理，就是在堆上模拟一个栈，只管分配，然后记住一个标记点一起释放。

又比如双端分配器，自己管理一大块内存，根据需求不同从两端向中间分配，这个还可以配合上面的分配器一起工作。

把对象的生命期绑定在渲染帧上，在一帧渲染完后把当帧的临时内存全部释放；或者做的更复杂一点，做一个乒乓开关，临时内存可以保留到下一帧结束。呵呵，这些以前都写过。

04 年的时候在网易公司内部做个一个比赛，就是在一块固定大小的内存内实现自己的内存管理器。用我们从梦幻西游，以及大话西游的客户端中实际采集来的数据做比赛评分，分别按允许速度和碎片率打分。我自己虽然没有参赛，但当时也写过一份程序拿到了最高分 :) 当时比赛的前三名现在都是网易项目（或离职后是新公司的主程）的主程。

现在关注的问题不太一样了，在 PC 平台上开发，一般就直接把 jemalloc 等成熟库拿来用，不太关注这部分的优化了。只在最近做 iOS 开发才重新和阿楠讨论过内存受限平台的内存管理问题。我们定制了一个 iOS 上供 Lua 用的特制分配器，用实际项目的数据采集结果做特定优化，效果还不错。

书中提到顽皮狗的引擎在做资源管理时，一律使用 handle 而不是直接内存地

址指针来索引对象。这样，所有被内存管理器管理的内存块都是可以被移动的。因为使用的是 handle 来做相互引用，就不存在指针重定位问题。

引擎可以利用空闲时间，慢慢做内存整理工作。把释放掉的内存块空间压实，做到内存碎片率为零。Cache 利用率也相当高。

如果所有的引擎模块都是自己维护，那么这么做相当棒，据说顽皮狗的引擎可以保证这一点。即使用了第三方库，谨慎选择这些库，通过定制分配器也可以适配的不错。

我以前想过这个做法，但没有实际试验过。书读到这里，突然有了点兴趣。Talk is cheap, show me the code．花了点时间，我写了段简单的代码实现了这么一个 Bump Pointer Allocator 。

这个库可以管理你指定的内存块，切割内存使用。所有内存块都用 id 索引，id 不会复用（在 32bit 用完前），所以你可以方便的知道一个 id 已经无效，这相当于支持了弱引用。从 id 映射到地址的操作是 O(1) 的，非常廉价，在显式执行整理操作前，内存不会移动。

内存分配也是 O(1) 的，只在少数情况下会退化成 O(n) 。释放操作只是减引用，也是 O(1) 的。内存整理工作可以定帧调用，虽然它需要 stop the world ，但可以保证在固定时间做完至少一步（除非有单个内存块特别大）。

内存块支持引用计数，由于有弱引用支持，所以使用者可以比较容易的回避循环引用的问题。

它还有许多改进空间：例如支持多线程；支持某些块禁止移动，方便外部保有指

针。

如果谁有兴趣可以加以改进。

原文

http://blog.codingnow.com/2013/11/bump_pointer_allocator.html?utm_source=Tuicool_Weekly

Python 项目自动化部署最佳实践

今天主要介绍下我们组刚刚开源出来的一个自动化部署的工具 essay，功能在 readme 上已经介绍的很详细了，这篇文章只是介绍下外围的情况，产生的环境，一些决策的考虑。

诞生之初

事情还得从头开始说起，从那些自动化的 fabric 文件开始，也从我刚入职搜狐负责手机搜狐开发开始说起。我参与开发的时候项目的部署已经是自动化了，不过并没有抽象出一个工具来。那会儿主要由两个项目，一个基于 tornado，一个基于 Django。两个项目都有各自的发版方法，但逻辑基本一致。

两个项目的上线流程都是先打包（py 的源码包），然后在通过内部的 pypiserver 安装到各个服务器上，由 supervisord 启动、管理。

随着业务的发展，新的项目逐渐多了起来。这时一个新的项目开发流程是这样的：

先从就项目中把 fabfile（fabric 的配置文件）和 supervisord 配置文件以及 setup.py 文件拷贝过来，然后再往里面填源码。流程依然是一致的。

这就是一开始的状态，混沌中带着那么一点秩序。

开始造轮子

说起来，程序员都是十足的懒人。这样 copy 的方式多少让自己都有点不忍直视，于是 @熊总 建议我们不如造这样一个轮子，让所有的项目都能在这上面滚起来。于是把这个造轮子的任务交给我来做，刚好我也是懒人一个，于是很 happy 的开始了。

要造一个通用的轮子，必然是要把项目中用到的部分抽象出来，哪些部分是通用的呢，这只有深切参与到项目的开发和部署中才能体会得到。刚好在那段时间之前，我也参与了修改 bug，打包，部署上线的过程，包括 copy 那些打包的脚本和配置文件。

有了上面的经验，只要把必需的东西输出就行了。总结了一下，当时项目的打包和上线涉及到这几个方面：：

1. 打包 —— 生成版本号，渲染 setup 中的版本和项目信息，然后放到 pypi server 的 packages 目录下
2. 虚拟环境 —— 在新的服务器上创建虚拟环境
3. 安装项目 —— 从 pypiserver 安装项目到虚拟环境中
4. 启动 supervisord —— 管理项目进程
5. 切换 nginx 配置 —— 我们有两套环境在线上同时运行，可以称为 a 环境和 b 环境，主要用于上线以及线上突然出现问题时回滚

细分的话就上面五个步骤，不太理解的可以去看看我们的 essay 说明。也就是只要满足了这些功能，那么这轮子就算是完成了。另外还考虑到为了便于新项目

的开发，还需要能自动创建具备这些功能的项目模板。这其实是最主要的痛点，总是拷贝什么的最没技术含量了。

于是添加了创建项目并且初始化模板，然后还能初始化到 gitlab 或者 github 上。这样的工具俨然是项目开发部署、居家旅行之良品。

争论之处

需求明确之后，怎么组织项目的代码呢，对于正常的 web 项目来说，没有啥难度的，都有固定的模板。对于这个工具类的东西，还是第一次考虑，怎么设计才能更合理——易扩展、易维护。在翻了好久 django 的源代码之后，我开始按照一个 core，然后一些 tools 的思路开始码代码。

我的考虑是，这个工具应该有一个核心的功能，然后是周围的一些辅助工具。遗憾的是，这种思路最后还是被推翻了。熊总的意思是应该参考 linux 的 pipeline 来设计，所有的功能都应能单独拿出来。于是底层的工具模块都按照这个逻辑被他重写了 (so，如果你们觉得那部分代码有槽点尽管吐好了，我不介意的，^_^)。

代码结构的争论还好些，基于经验就能看出哪种更好。但是一些逻辑的设计却不是经验能得到的，就像软件开发没有银弹一样，各自的业务场景都不同，没有统一的解决方案。

另外一个争执的点是部署方式。如果你已经看了我们的文档，或者已经理解了上面的部署方式。你可能已经疑惑了：“为毛你们不直接用 git 部署呢?还可以打 tag 什么的。” 这也是我们之前在考虑的问题。

摆在我们面前的有两条路，一条路是用 git 来部署代码，另外一条路是用 pip install 项目包来部署。我们选择了后者。原因是这样的：：

> 1. 历史原因 —— 之前的项目一直在用这样的方式
>
> 2. 服务器配置的成本 —— 这个我觉得是最主要的，对比两种方式，git 部署的话服务器要统一安装 git 环境，但是我们申请到新的服务器没有这东西，我们得自己安装；另外还有一个包依赖的问题。而使用 pip 的方式安装，不需要做多余的处理，新来机器，给了 ip，直接就能部署上去。

大概就基于上述的两个原因，选择了用 pip install 的方式来部署了。有什么没说到得地方，有同事路过留言补充下吧。

开放的初衷

有了上面的过程，也就产生了这么个工具： essay 。在项目完成之后，后面的时间里我们划分了不同的组，又开始负责新的项目。这个工具算是一直在我们的项目开发中起着重要的作用。我们觉得这个工具算是我们在过去一年多中开发和部署经验的总结，开放出来应该会有些价值，无论是对于开发者还是对于团队。我自己是在这个工具的开发过程中学到很多东西，我想任何一个渴望了解项目从开发到部署整个流程的开发人员都应该能从中有所收益。

开放的目的除了分享经验，还有一个重要的作用就是交流。我们所积累的经验在业内并不一定是最佳的，肯定还有更多更好的解决方案，而这些东西都要来源于交流。这样才能相互促进，而相互促进才是开放和开源的初衷。

项目地址：https://github.com/SohuTech/essay

补充一些数据

手机搜狐网 (m.sohu.com)每天有几亿的 PV，在这样的情况下，发现线上 bug，一般情况下从修复 bug 到上线不会超过 15 分钟。并且在上线的时候用户是不会感觉到页面访问慢或者打不开的。在新功能点或者 bug 多的情况下，一天上线十几个版本的情况也是有的。每次上线都不会对用户造成影响的关键在于我们部署了 a，b 两套环境。

之前在我参与手机搜狐网开发时后台有 100 多台虚机，在遇到热点事件的时候会扩充一倍。在这种情况下，新功能上线的过程我印象中不会超过 5 分钟，如果关闭 fabric 的 success 的 console 输出以及开启并行模式，发布的速度会更快。

原文

http://www.the5fire.com/auto-deploy-tool-for-python-app.html?utm_source=Tuicool_Weekly

Java SE 8: 标准库增强

Lambda 表达式是 Java SE 8 的核心功能，大部分的改进都围绕 lambda 表达式展开。（Jigsaw 项目已经被推迟到 Java SE 9。）关于 lambda 表达式的内容，已经在上一篇文章中进行了说明。这篇文章主要介绍 Java SE 8 中包含的其他 Java 标准库的增强。

并行排序

随着多核 CPU 的流行，Java 平台的标准库实现也尽可能利用底层硬件平台的能力来提高性能。Java SE 7 中引入了 Fork/Join 框架作为一个轻量级的并行任务执行引擎。Java SE 8 把 Fork/Join 框架用到了标准库的一些方法的实现中。比较典型的是 java.utils.Arrays 类中新增的 parallelSort 方法。与已有的 sort

方法不同的是，parallelSort 方法使用 Fork/Join 框架来实现。在多核 CPU 平台上的性能更好。下面的代码对包含 1 亿个整数的数组分别使用 parallelSort 和 sort 进行排序。

```
Random random = new Random();
int count = 100000000;
int[] array = new int[count];
Arrays.parallelSetAll(array, (index) -> random.nextInt());
int[] copy = new int[count];
System.arraycopy(array, 0, copy, 0, array.length);
Arrays.parallelSort(array);
Arrays.sort(copy);
```

在本人的 4 核 CPU 的平台上，parallelSort 和 sort 方法的耗时分别是 7112 毫秒和 16777 毫秒。所以 parallelSort 方法的性能要好不少。不过 parallelSort 方法只在数据量较大时有比较明显的性能提升。当数据量较小时，Fork/Join 框架本身所带来的额外开销足以抵消它带来的性能提升。

集合批量数据操作

在 Java 应用的开发中，对集合的操作是比较常见的。不过在 Java SE 8 之前的 Java 标准库中，对集合所能进行的操作比较有限，基本上都围绕集合遍历来展开。相对于其他编程语言来说，Java 标准库在这一块是比较弱的。Java SE 8 中 lambda 表达式的引入以及标准库的增强改进了这种状况。具体来说体现在两个方面上的改进：第一个方面是对集合的操作方式上。得益于默认方法的引入，Java 集合框架中的接口可以进行更新，添加了更多有用的操作方式，即通常所说的“filter/map/reduce”等操作。第二个方面是对集合的操作逻辑的表示方式上。新添加的操作方式使用了 java.util.function 包中的新的函数式接口，可以很方便地使用 lambda 表达式来表示对集合的处理逻辑。这两个方面结合起来，得到的是更加直观和简洁的代码。

新的集合批量处理操作的核心是新增的 java.util.stream 包，其中最重要的是 java.util.stream.Stream 接口。Stream 接口的概念类似于 Java I/O 库中的流，表示的是一个支持顺序和并行操作的元素的序列。在该序列上可以进行不同的转换操作。序列中包含的元素也可以被消费以产生所需的结果。Stream 接口所表示的只是操作层面上的抽象，与底层的数据存储并没有关系。通常的使用方式是从集合中创建出 Stream 接口的对象，再进行各种不同的转换操作，最后消费操作执行的结果。

Stream 接口中包含的操作分成两类：第一类是对序列中元素进行转换的中间操作，如 filter 和 map 等。这类中间操作是延迟进行的，可以级联起来。第二类

是消费序列中元素的终止操作，如 forEach 和 count 等。当对一个 Stream 接口的对象执行了终止操作之后，该对象无法被再次处理。这点符合一般意义上对于“流”的理解。下面的代码给出了 Stream 接口中的 filter、map 和 reduce 操作的基本使用方式。Stream 接口中的方法大量使用了函数式接口，可以用 lambda 表达式很方便地进行操作。

```
IntStream.range(1,10).filter(i -> i % 2 ==
0).findFirst().ifPresent(System.out::println);
//保留偶数并输出第一个元素
```

```
IntStream.range(1,10).map(i -> i * 2).forEach(System.out::println);
//所有元素乘以 2 并输出
int value = IntStream.range(1, 10).reduce(0, Integer::sum); //求和
```

Stream 接口的 reduce 操作还支持一种更加复杂的用法，如下面的代码所示：

```
List<String> fruits = Arrays.asList(new String[] {"apple", "orange", "pear"});
int totalLength = fruits.stream().reduce(0, (sum, str) -> sum + str.length(),
Integer::sum);
//字符串长度的总和
```

这种方式的 reduce 方法需要 3 个参数，分别是初始值、累积函数和组合函数。

初始值是 reduce 操作的起始值；累积函数把部分结果和新的元素累积成新的部分结果组合函数则把两个部分结果组合成新的部分结果，最后产生最终结果。这种形式的 reduce 操作通常可以简化成一个 map 操作和另外一个简单的 reduce 操作，如下面的代码所示。两种方式的效果是一样的，不过下面的方式更加容易理解一些。

```
int totalLength = fruits.stream().mapToInt(String::length).reduce(0,
Integer::sum);
```

另外一种特殊的 reduce 操作是 collect 操作。它与 reduce 的不同之处在于，collect 操作的过程中所进行的是对一个结果对象进行修改操作。这样可以避免不必要的对象创建，提高性能。下面代码中的结果是一个 StringBuilder 类的对象。

```
StringBuilder upperCase = fruits.stream().collect(StringBuilder::new,
(builder, str)
-> builder.append(str.substring(0, 1).toUpperCase()),
StringBuilder::append);
//字符串首字母大写并连接
```

Stream 接口中的操作可以是顺序执行或并行执行的。这是在 Stream 接口的对象创建时所确定的。比如 Collection 接口提供了 stream 和 parallelStream 方法来创建两种不同执行方式的 Stream 接口的对象。这两种不同的方式是可以切

换的，通过 Stream 接口的 sequential 和 parallel 方法就可以完成。

日期和时间

Java 标准库中的日期和时间处理 API 一直为开发人员所诟病。大多数开发人员会选择 Joda Time 这样的第三方库来进行替代。JSR 310 作为 Java SE 8 的一部分，重新定义了新的日期和时间 API，借鉴了已有第三方库中的最佳实践。I 定义在 java.time 包中的新的日期和时间 API 基于标准的 ISO 8601 日历系统。在新的日期和时间 API 中，核心的类是 LocalDateTime、OffsetDateTime 和 ZonedDateTime。LocalDateTime 类表示的是 ISO 8601 日历系统中不带时区的日期和时间信息。OffsetDateTime 类在基本的日期和时间基础上增加了与 UTC 的偏移量。ZonedDateTime 类则加上了时区的相关信息。下面的代码给出了日期和时间 API 的基本用法，包括对日期和时间的修改、输出和解析。

```
LocalDateTime.now().plusDays(3).minusHours(1).format(DateTimeForm
atter
.ISO_LOCAL_DATE_TIME); //输出日期和时间
ZonedDateTime.now().withZoneSameInstant(ZoneId.of("GMT+08:00")).f
ormat
(DateTimeFormatter.ISO_ZONED_DATE_TIME); //输出日期、时间和时区
DateTimeFormatter.ofPattern("yyyy MM dd").parse("200101 25").
query(TemporalQuery.localDate()); //日期的解析
```

除了上述 3 个类之外，还有几个值得一提的辅助类。

- Instant：表示时间线上的一个点。当程序中需要记录时间戳时，应该使用该类。Instant 类表示的时间类似“2013-07-22T23:18:35.743Z“。
- Duration：表示精确的基于时间的间隔。比如 Duration.of(30, ChronoUnit.SECONDS)可以获取 30 秒的间隔。需要注意的是，Duration 类的对象只能从精确的时间间隔创建出来，如秒、小时和天等。在这里，一天表示精确的 24 小时。而月份和年是不能使用的，因为它们不能表示精确的间隔。
- Period：与 Duration 类相对应的 Period 类表示的是基于日期的间隔，只能使用年/月/日作为单位。Period 类在计算时会考虑夏令时等因素，适合于计算展示给最终用户的内容。
- Clock：表示包含时区信息的时钟，可以获取当前日期和时间。如 LocalDateTime.now(Clock.system(ZoneId.of("GMT+08:00")))表示的是当前的北京时间。Clock 类的一个重要作用是简化测试。在测试时可以指定不同时区的时钟来进行模拟。

其他更新

除了上面提到的几个比较大的更新之前，还有一些小的改动。

Base64 编码

Base64 编码在 Java 应用开发中经常会用到，比如在 HTTP 基本认证中。在 Java SE 8 之前，需要使用第三方库来进行 Base64 编码与解码。Java SE 8 增加了 java.util.Base64 类进行编码和解码。下面的代码给出了简单的示例。

```
Base64.Encoder encoder = Base64.getEncoder();
String encoded =
encoder.encodeToString("username:password".getBytes());
Base64.Decoder decoder = Base64.getDecoder();
String decoded = new String(decoder.decode(encoded));
```

并发处理

Java SE 8 进一步增强了并发处理的相关 API。在 java.util.concurrent.atomic 中新增了 LongAccumulator、LongAdder、DoubleAccumulator 和 DoubleAdder 等几个类。这几个类用来在多线程的情况下更新某个 Long 或 Double 类型的变量。下面的代码给出了 LongAccumulator 类的使用示例。

```
public class ConcurrentSample {
    public static void main(String[] args) throws Exception {
        ConcurrentSample sample = new ConcurrentSample();
        LongAccumulator accumulator = new
LongAccumulator(Long::max,
Long.MIN_VALUE);
        for (int i = 0; i < 100; i++) {
            sample.newThread("Test thread - " + i, accumulator);
```

```
            }
            System.out.println(accumulator.longValue());
        }

        private void newThread(final String name, final LongAccumulator
accumulator)
  throws Exception {
            Thread thread = new Thread(() -> {
                Random random = new Random();
                int value = random.nextInt(5000);
                System.out.println(String.format("%s -> %s",
Thread.currentThread
().getName(), value));
                accumulator.accumulate(value);
            }, name);
            thread.start();
            thread.join();
        }
```

```
}
```

当 LongAccumulator 类的对象上的 accumulate 方法被调用时，参数中的值会通过 Long 类的 max 方法进行比较，所得到的结果作为 LongAccumulator 类的对象的当前值。经过多次累积操作之后，最终的结果是所有调用操作中提供的最大值。

ConcurrentHashMap 类得到了比较大的更新，添加了很多实用的方法，如 compute 方法用来进行值的计算，merge 方法用来进行键值的合并，search 方法用来进行查找等。这使得 ConcurrentHashMap 类可以很方便的创建缓存系统。

原文

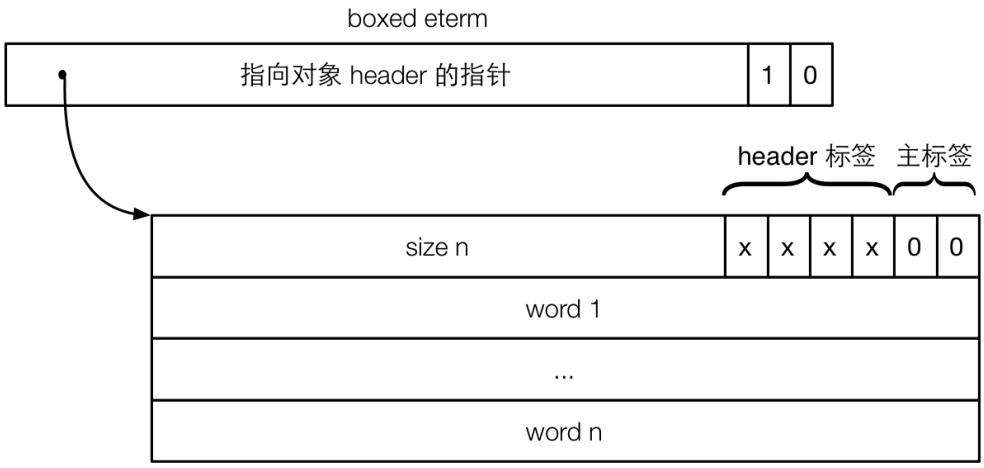
http://www.infoq.com/cn/articles/java-se8-standard-library-enhancements?utm_source=Tuicool_Weekly

Erlang 数据类型的表示和实现（4）——boxed 对象

Boxed 对象

Boxed 对象是比较复杂的对象，在 Erlang 中主标签为 10 的 Eterm 表示一个对 boxed 对象的引用。这个 Eterm 除去标签之后剩下的实际上是一个指针，指向具体的 boxed 对象。如下图所示，boxed 对象由对象头和具体的数据组成，这些字都排布在一起，占用进程栈中的一段连续空间（不像列表那样会

分开）。



对象头分为 3 部分：主标签固定为 00（因此也没有 Eterm 以 00 为主标签），然后是 4 个位表示的 header 标签，这个标签表示了这个 boxed 对象的具体类型。接下来剩下的部分就是对象的大小 n，在对象头之后的 n 个字就是这个对象的具体数据。具体数据的格式和意义取决于具体的对象类型。对象头中表示大小的值也称为对象的 arity，目前在 Erlang 虚拟机中规定最多使用 24 位表示这个 n，因此对象最大不超过 224=16777215 个字。

下面列出了 Erlang 虚拟机支持的所有 header 标签：

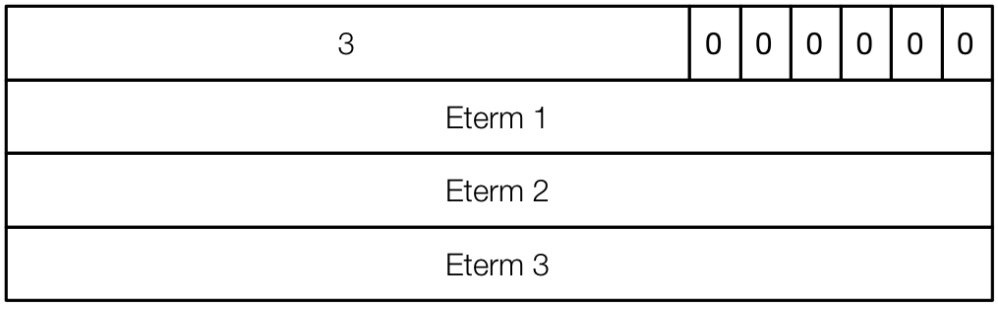
- 0000：表示元组，高位的 size 即元组中元素个数。
- 0001：内部使用的 binary 匹配状态。
- 001x：大数（bignum），x 为符号位，可以表示任意大的整数，限制取决于内存。
- 0100：本地 ref
- 0101：fun
- 0110：浮点数

- 0111：export 信息
- 1000：refc_binary，引用计数的 binary，即大 binary
- 1001：heap_binary，小 binary，直接放在堆中
- 1010：sub_binary，分离 binary 时产生的子 binary
- 1011：没有使用
- 1100：外部 pid
- 1101：外部 port
- 1110：外部 ref
- 1111：没有使用

以上这些 boxed 对象在 Erlang 虚拟机内都称为 thing。这些“东西”有一些是表示 Erlang 开发者可以直接使用的数据类型，有一些则表示内部数据类型。下面我们来依次了解这些数据类型。

元组

元组很简单，实际上就是一个数组，如下图所示的一个 3 元素的元组：



元组中的元素在内存中依次排开，中间没有间隔，每一个元素都是一个 Eterm，所以元组中的元素可以是任意类型的。和列表一样，如果创建一个元组的时候引用了其他对象，那么这些被引用的对象也是共享的。但是当元组跨越了进程的边

界的时候，也会被扁平化。

大整数，浮点数

大整数的符号保存在对象头的标签中，因此大整数对象本身的数据保存的就是大整数的绝对值。由于 Erlang 支持任意大的整数，所以大整数的长度一定是至少 1 个字的。那么大整数是以什么样的格式保存在这些机器字中的呢？假设某个大整数需要用 n 个字表示，而且每一个机器字宽度为 64 位元，那么这个大整数的值为：

$S=(264)n-1\times wn-1+(264)n-2\times wn-2+\cdots+(264)0\times w0=\sum i=0n-1(264)i\times wi$

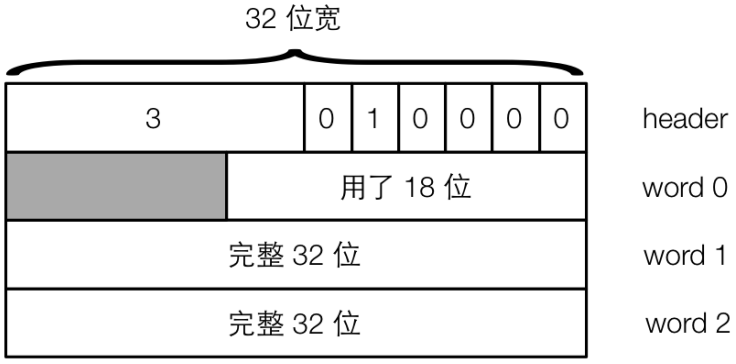
其中 *wi* 表示第 *i* 个机器字。可以看出，低地址处的机器字表示大整数中的低位。实际上，就是从高地址到低地址一段一段拼起来。Erlang 虚拟机在操作大整数的时候，计算方法和我们笔算算数是一样的。我们做算数笔算的时候，是一个数字一个数字地挨个计算的，如果碰到进位，则加到更高一位的数字中。Erlang 虚拟机的大数算法在做计算的时候，也是一个数字一个数字地算，只不过刚好使用一个机器字作为我们笔算时的一个数字，在虚拟机的代码中，也是把一个机器字 typedef 为 ErtsDigit。换句话说，我们笔算是在算 10 进制数，而 64 位 Erlang 虚拟机内部则是在进行 264 进制的整数计算。Erlang 大数计算的算法都在 erts/emulator/beam/big.c 文件中，有兴趣的读者可以读一读，虽然原理挺简单，但是里面还是有不少技巧的。比如说 Erlang 虚拟机在进行大数计算之前，要事先为结果分配好空间，分配好空间之后大数对象的大小就固定了，大数算法在计算的过程中将结果填在分配好的空间中。此外还有不少大数到字符串（各种进制的表达）之间的转换，在这种转换过程中需要在两个方向预估对方数

据类型所需的空间大小。Erlang 虚拟机采用了信息熵的方法，通过查表的方式查找某个进制下的一个数码（应该是 digit，但是在代码中由于已经用 digit 表示一个固定机器字，所以 big.c 代码中用了 character 这个词表示大整数的一个数码）所需的位元数（即 log2*r*，*r* 表示进制）。

浮点数就比较简单了。Erlang 中的浮点数是双精度浮点数，实际上就是编译器原生的 double 类型，因此在 64 位的机器上刚好是一个机器字的大小。那么浮点数对象的 header 很简单，除去固定标签之外，arity 值固定为 1。然后后面跟着的一个字大小的数据就是浮点数符合 IEEE 754 规范的表示形式（注 4）。

本地 ref

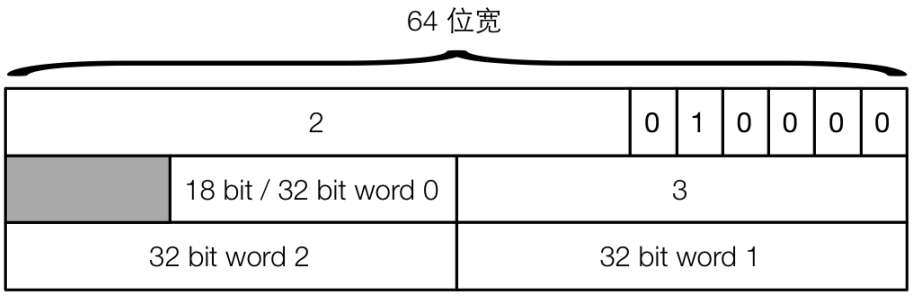
如果是在 32 位机器上，ref 对象表示形式如下所示：



header 里面的 arity 总是设置为 3，后面跟着 3 个 32 位的机器字，其中第 0 个只用了 18 位。这个图也结解释了为什么 ref 的值超过了 282 之后就会绕回为 0（18+32+32=82）。

在 64 位的机器上，ref 能表示的数据宽度也是 82 位，如果和 32 位机器不一样的话就不好交换数据了。那么如果在 64 位的机器上，只是简单地拓宽

header 以及后面的 3 个字，那么每个 ref 就要浪费 4×3=12 个字节的空间，所以在小尾顺序的 64 位机器上，就成下面这样子了：



其中 word 0 依然只使用了 18 位，而且在 header 后面的第一个机器字的低 32 位还保存了一个数字 3，即后面跟着的 32 位元字数。

Erlang 虚拟机通过 3 个全局变量分别表示 word 0、1 和 2 的当前值。调用 BIF make_ref() 的时候，Erlang 虚拟机会创建一个新的 ref。由于全局当前 ref 值是用多个变量表示的，所以 make_ref() 会通过一个自旋锁保护对这些变量的操作，递增全局 ref 的值，然后根据新的 ref 值创建新的 ref 对象并返回对应的 Eterm。递增操作针对 word 0 递增，如果 word 0 超过了 218，则进位到 word 1，word 1 归零的话则进位到 word 2。

我们打印 ref 的时候，得到的是类似 #Ref<0.0.0.2055> 这样的输出，通过 3 个句点将输出结果分为 4 段。第 1 段，和 pid 和 port 的第一段是一样的，表示节点，在本地节点总是为 0，后面 3 段分别为上面的 word 2、1 和 0。所以 ref 较少的时候前面几段都为 0。

binary

binary 的内容比较繁多，单独放在本系列的第 5 篇了。

外部 pid、port 和 ref

外部 Eterm 是 Erlang 虚拟机的分布式节点之间交换 Eterm 时所用的格式。

由于具体涉及到分布式 Erlang 的工作细节，所以打算在专门介绍 Erlang 分布式机制的博文中具体讨论。

fun 和 export

涉及到 Beam 虚拟机的代码格式和工作原理，打算放在专门介绍 Beam 虚拟机的博文中具体讨论。

原文
http://www.cnblogs.com/zhengsyao/p/erlang_eterm_implementation_4_boxed.html?utm_source=Tuicool_Weekly

精简自己 20%的代码

一：发现问题

先来说如何重构业务层的 try{}catch{}finally{}代码块，我看过很多代码，异常处理这一块大致分为两种情况，一种是每个方法都大量的充斥着 try{}catch{}finally{}，这种方式的编程已经考虑到了异常处理，还有一种就是没有 try{}catch{}finally{}的代码，因为根本就没有考虑代码的异常处理。每当我看到这样的代码，我都很忧伤。从程序的健壮性来看第一种还是要比第二种情况好，至少在编程意识中，随时想到了异常情况，有一种基本的编程思想。

比如：一个业务单据的多表插入，关联修改，虚拟删除等都是一些基本的操作，但是又是比较容易引起错误的操作，在这些方法上都会加上 try{}catch{}finally{}对代码进行有效的防错处理。早期的代码是这样的。

```
public Boolean Save(AccountModel accountData)
{
    Boolean result = false;
    try
    {
        //TODO ...
        result = true;
    }
    catch
    {
```

```
    }
    finally
    {

    }
    return result;
}

public Boolean Edit(AccountModel accountData)
{
    Boolean result = false;
    try
    {
        //TODO ...
        result = true;
    }
    catch
    {

    }
    finally
    {

    }
    return result;
}

public Boolean VirDelete(AccountModel accountData)
{
    Boolean result = false;
    try
    {
        //TODO ...
        result = true;
    }
    catch
    {

    }
    finally
    {

    }
    return result;
```

```
}
```

仅仅定义了添加，修改，删除几个空方法，就写了三四十行代码，如果业务稍微复杂些，异常处理的代码很快就会突破百行大关。虽然复制，粘贴try{}catch{}finally{}很好使，但是业务逻辑代码大量充斥着这样的try{}catch{}finally{}代码，确实显得做事不够利落。

二：解决问题

那怎样来解决这件棘手的事呢，首先定义一个公用的try{}catch{}finally{}，如下如示：

```
public class Process
{
    public static bool Execute(Action action)
    {
        try
        {
            action.Invoke();
            return true;
        }
        catch (Exception ex)
        {
            //1，异常隐藏
            //2，异常替换
            //3，异常封装

            //写日志
            return false;
        }
        finally
        {

        }
    }
}
```

上边的代码定义了公用的try{}catch{}finally{}，最关键是怎么运用起来，如下代码：

```
protected void Page_Load(object sender, EventArgs e)
{
    AccountModel accountData = new AccountModel();      //准备传入的参
```

```
数
    Boolean result = false;                                  //接收返回的值
    Process.Execute(() => result = Save(accountData)); //执行方法
}

public Boolean Save(AccountModel accountData)
{
    Boolean result = false;
    //TODO ...
    result = true;
    return result;
}

public Boolean Edit(AccountModel accountData)
{
    Boolean result = false;
    //TODO ...
    result = true;
    return result;
}

public Boolean VirDelete(AccountModel accountData)
{
    Boolean result = false;
    //TODO ...
    result = true;
    return result;
}
```

这样的精简过的代码，是不是感觉心情很舒畅。

三：提升与扩展

对于知足者常乐的人来说，到第二个步骤就可以洗洗睡了。但是对于精益求精的人来说，问题仍然没有完。我们来说一个应用场景，在 WCF 中的应用，我们知道 WCF 服务端的异常，不经过<serviceDebug includeExceptionDetailInFaults="true"/>的设置，服务端的异常是无法抛到客户端的。但是在正式环境中，不可能对进行 serviceDebug 的配置。正确的处理是在服务端对异常进行隐藏，替换，或者封装。

比如我们在服务端捕获了一个已知异常，但是这个异常会暴露一些敏感的信息，所以我们对异常进行替换，抛出新的异常后，我们还要把这个异常怎样传输给客户端。首先们要明确 WCF 中的一些基本常识，就是 WCF 中的数据传递要遵循 WCF 的数据契约，服务端抛到客户端的异常（异常其实也是数据），所以必须要给异常定义异常契约。

```
[DataContract(Name = "WCFException")]
public class WCFException
{
    [DataMember(Name = "Type")]
    public String Type { get; set; }

    [DataMember(Name = "StackTrace")]
    public String StackTrace { get; set; }

    [DataMember(Name = "Message")]
    public String Message { get; set; }
}
```

然后处理异常的公共方法改写为：

```
public static bool Execute(Action action)
{
    try
    {
        action.Invoke();
        return true;
    }
    catch (Exception ex)
    {
        //1，异常隐藏
        //2，异常替换
        //3，异常封装

        //写日志
        WCFException exception = new WCFException
        {
            Type = "Error"
            ,
            StackTrace = ex.StackTrace
            ,
            Message = ex.Message
        };
        throw new FaultException<WCFException>(exception
            , new FaultReason("服务端异常:" + ex.Message)
            , new FaultCode(ex.TargetSite.Name));
    }
    finally
```

```
    {

    }
}
```

这样在服务端抛出的异常，就能在客户端捕捉到。现在是不是感觉自己又提升了一些，想成为编程高手是指日可待了。

四：举一反三

异常的处理也不过如此，那是不是应该举一反三，看看事务的处理应该怎么办？比如现在大量的访求都用到了事务，如下代码：

```
public Boolean Save(AccountModel accountData)
{
    OracleConnection conn = new OracleConnection("连接字符串");
    IDbTransaction trans = conn.BeginTransaction();
    Boolean result = false;
    try
    {
        //TODO ...
        trans.Commit();
        result = true;
    }
    catch
    {
        trans.Rollback();
    }
    finally
    {

    }
    return result;
}
```

特别是 trans.Commit(); trans.Rollback(); 这样的代码出现在每个与事务相关的方法中， 让我感觉到代码的臃肿，以及隐陷约约的失望。
经过我几天的翻阅资料终于实现了事务的公用访求提取。使用方法如下代码所示：

```
[TransactionAttribute]
[ExceptionAttribute]
public bool Save(DataContext dContext, Dictionary<string, string>
```

```
dtoPara)
{
    Boolean returnVal = true;
    //TODO ...
    return returnVal;
}
```

就是在一个方法上加[TransactionAttribute]就表示这个方法写在了事务中，反之，不在事务中，加[ExceptionAttribute]就表示这个方法作了异常处理，反之，不作异常处理。通过反射或者 AOP 都能实现 Attribute 编程的效果。

原文

http://www.cnblogs.com/xcj26/p/3442089.html?utm_source=Tuicool_Weekly

Android 应用性能 分析

其实主要是内存方面，内存管理是个永恒的话题!

1.从工具 DDMS 中，在 Sysinfo 的 tab 栏里面有一个 Memory usage 的选项，通过 USB 连接 Android 设备以后很容易抓到图。

在图中可以看到系统随时可以用的内存是 Free 和 Buffers 两项，因为我抓图的系统只有 128M 的内存，所以看上去这部分可用内存已经很少了。

2.通过 Linux 的/proc 文件系统的 meminfo 来分析这个系统的内存使用情况更客观。之所以这么说，是因为通过这种方法可以绕开繁琐的 dalvik 实现机制，以系统的层面来分析:

C:\Users\Administrator>adb shell

shell@android:/ $ cat /proc/meminfo

cat /proc/meminfo

MemTotal: 999008 kB

MemFree: 157532 kB

Buffers: 41308 kB

Cached: 319584 kB

SwapCached: 0 kB

Active: 488128 kB

Inactive: 167012 kB

Active(anon): 292356 kB

Inactive(anon): 3544 kB

Active(file): 195772 kB

Inactive(file): 163468 kB

Unevictable: 1520 kB

Mlocked: 13684 kB

HighTotal: 529408 kB

HighFree: 61680 kB

LowTotal: 469600 kB

LowFree: 95852 kB

SwapTotal: 0 kB

SwapFree: 0 kB

Dirty: 0 kB

Writeback: 0 kB

AnonPages: 295816 kB

Mapped: 165768 kB

Shmem: 360 kB

Slab: 27752 kB

SReclaimable: 9524 kB

SUnreclaim: 18228 kB

KernelStack: 8232 kB

PageTables: 9628 kB

NFS_Unstable: 0 kB

Bounce: 0 kB

WritebackTmp: 0 kB

CommitLimit: 499504 kB

Committed_AS: 5690380 kB

VmallocTotal: 540672 kB

VmallocUsed: 139020 kB

VmallocChunk: 98112 kB

shell@android:/ $

对于 Linux 系统来说，可以立即使用的内存是 MemFree+Buffers+Cache，

我们从 DDMS 中拿到的图差很多。或者说 Google 隐藏了 cache，没有给我我想要的东西；Android 系统为了加快系统的运行速度会在系统允许的情况下，大量的使用内存作为应用程序的 cache。而当系统内存紧张的时候，会首先释放 cache 的内存，

3、Android 内存介绍：

在 java 开发过程中，是通过 new 来为对象分配内存的，而内存的释放是由垃圾收集器（GC）来回收的，在开发的过程中，不需要显式的去管理内存，java 虚拟机会自动帮我们回收内存。但是这样有可能在不知不觉中就会浪费了很多内存，最终导致 java 虚拟机花费很多时间去进行垃圾回收，更严重的是造成 JVM 的 OOM。

4、APP 占用的内存分哪些：

Android 系统中的内存和 linux 系统一样，存在着大量的共享内存。每个 APP 占内存会有私有和公共的两部分：ShareDirty、PrivateDirty。Pss 是考虑共享内存的内核计算尺度 — 基本上一个进程的每个内存页面被按一个比率缩减，这个比率和同样使用该页面的其他进程的数量有关。理论上你可以累计所有进程的 Pss 占用量来检查所有进程的内存占用量，也可以比较进程的 Pss 来大致发现进程各自的权重。PrivateDirty，它基本上是进程内不能被分页到磁盘的内存，也不和其他进程共享。

手机中系统设置里有可以查看正在运行的应用程序所占的内存，此处显示的内存为该进程所占用的 Total Pss。所以我们只需要查看 Total Pss 的值就可以知道该应用运行时所占的内存的大小。

5、如何查看一个 APP 占用的内存，查看内存大致上有三种方法：

1．通过系统设置查看

在系统设置中->应用->正在运行->APP

优点：操作简单

缺点：数值不准确，无法实时查看数值变化

2. 通过命令行查看

adb shell dumpsys meminfo yourpakagename

其中 Pss 对应的 TOTAL 值为内存所实际占用的值

优点：简单方便，数据全面精确

缺点：无法实时查看内存占用

3. 通过系统 API 查看

首先通过 activitymanager 获得正在运行的程序列表，找到所要获取的程序的 pid。

activitymanager.getRunningAppProcesses()

再通过 memoryinfo[0].getTotalPss();方法获得实际内存占用

Memoryinfo 中还包括 getTotalPrivateDirty 和 getTotalSharedDirty 方法

优点：可拓展性高，可以通过程序实时查看内存的占用；数据全面精确

缺点：需要具有开发能力，入手较为困难，所以我们现在在测试内存的时候，是使用了内部自己研发的一款 APP 来监测内存的，这个 APP 目前可以实现实时监测并记录数据结果，可以提供给开发者和测试者分析内存的数据支持。目前仍然属于内测阶段，以后有机会可以提供给大家使用。

四、内存泄漏

何为内存泄漏？内存泄漏也称作“存储渗漏”，用动态存储分配函数动态开辟的空间，在使用完毕后未释放，结果导致一直占据该内存单元，直到程序结束。

内存泄漏的实例：

```
btn.setOnClickListener(new OnClickListener() {
@Override
```

```
public void onClick(View arg0) {
    for (int i = 0; i < 100; i++) {
    ImageView img = new ImageView(MainActivity.this);
    img.setImageResource(R.drawable.ic_launcher);

    test.add(img);
   }
 }
});
```

其中 test 为静态的 List<ImageView>。这样，如果一直点击 btn 就会出现内存泄漏的情形。

我们如何去监测内存泄漏呢?

以上面内存泄漏的例子为测试 Activity，当点击按钮后，会向一个静态数组中添加图片，这样就形成了一个内存泄漏的场景。

进入测试 Activity，在点击按钮前先记录当前 APP 所占用的内存，然后点击按钮。等待操作执行完成后，进行一次 GC，再查看 APP 所占用的内存。

返回后 APP 所占用的内存没有明显的回落，表明在代码中可能存在内存泄漏的情况发生。

即：在执行某种操作后进行一次 GC，内存没有明显的回落。此时即可以断定代码中可能存在内存泄漏。

检测方法：

通过上文所用的三种方法去查看内存的使用情况

使用 DDMS 中的 Heap：

1)　　打开 DDMS 并打开 Devices 视图和 Heap 视图

2)　　点击选择要监控的进程

3)　　选中 Devices 视图界面上的”update heap”　图标

4)　　点击 Heap 视图中的”Cause GC”　按钮（相当于进行了一次 GC 的操作）

一般我们会观察 Data Object 的 Total 值，正常情况下在每次 GC 后，这个值都会有明显的回落并会稳定在一个范围之内，说明代码中没有未被释放的内存；若这个值在每次 GC 后没有出现明显的回落，则说明代码中可能存在没有被释放的内存。

总述：内存不仅是性能测试时需要关注的，作为优秀的开发工程师更应该关注自己的代码内存占用的情况，这样可以尽量避免 OOM 的情况发生。要知道手机分配给每个进程的内存并不多，当系统内存不够的时候会 kill 掉一些占内存高的进程，所以为了不被系统 kill 掉我们要尽可能的合理使用内存避免内存泄漏的情况发生。

原文

http://blog.csdn.net/lixiaopeng23/article/details/16982843?utm_source=Tuicool_Weekly

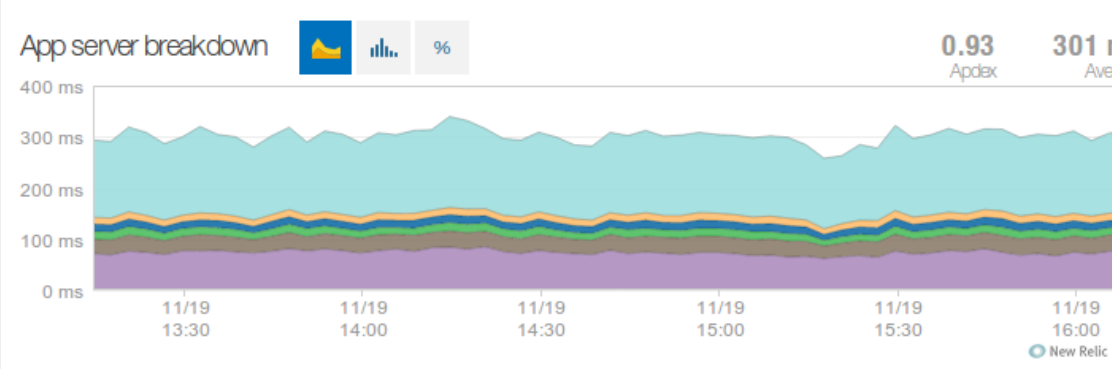
Rails 3.2 性能： 更慢了？

拥有一个大型代码库意味着我们不能很经常升级 Rails 的版本（我们平均每两年一次升级，每次升级需要 1-2 周的开发时间）。不过每次我们做升级工作的时候，

我最先好奇的事情之一是，检查不同版本之间的性能差异。

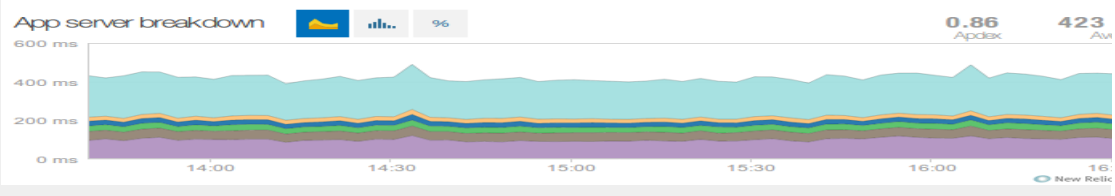
就我们之前的升级来说，在从 Rails 2.3 到 Rails 3.0 的过程中，我记录下来的平均动作变得要慢 2 倍，一个动作需要的平均时间由 225ms 攀升到 480ms。幸运的是，在这种情境下我们可以拿出一些技巧（GC 调优），这样我们最终将同样的动作时间缩减到 280ms。即使是实现一些新奇的技巧，这个时间仍然比 Rails 2.3 要慢约 25%，但是我们已经可以接受这个结果了。

当我们最终决定由 Rails 3.0 升级到 3.2，以便与最新的 gems 兼容的时候，由于我们之前的经历，可想而知我对性能有何等下降是多么的急切。根据手头现有数字，看来有必要对此加以理解。这里是上次我描述到的同一个动作（最常见的动作——显示一个条目的动作）的不同表现情况，在升级以前的 Rails 3.0 上：



在升级以前最常见的动作，在 3 小时的时间窗口中平均需要 301ms

这里是现在的样子：



升级以后，与上周相同时间段，在 3 小时的时间窗口中平均需要 423ms

与上次不同，3.2 的问题在于，我们再也没有办法凭空捏造更多的技巧。我们已经升级到最新最伟大的 Ruby 2.0 版本。在请求期间我们已经禁止了 GC（感谢有 Passenger！）。我们完成这些升级以后，Rails 3.0 的应用速度快了大约 25%。但是现在由于 Rails 3.2 中我们需要承受控制器与视图渲染的 40%性能下降，这些性能提升被遮蔽了，而且在做 Ruby 优化以前，反而比 3.0 中的速度还要慢。

总而言之，如果你有基于 Rails 的大型应用，此刻你可能已经懂得对 Rails 新版本心存畏惧。我完全理解那些为 Rails LTS 交钱的人。（译注：LTS 即 Long Term Support）如果我们不需要与新的 gems 兼容，仍然使用 2.3，这将使我们比 Rails 3.0 速度快 100%，相应的比 Rails 3.2 速度快 40%。

新版的 Rails 鼓吹各种改进，像“创建单页 web 应用的能力”、“更严格的默认安全性”以及“改革，简化”其组成库。我们看到的最近一次的性能提升，是 3.2 版本使开发环境的加载加快了(1)。这显然是一个令人难以置信的提升（使平均的开发页加载由 5 秒以上缩减到 1-2 秒），尽管由于 active_reload，我们已经在 Rails 3.0 有这样的性能提升。

我觉得在驱动 Rails 开发的各种要素中，现在性能已经成为最不被关注的一个了，如果真是如此，那么这是令人相当羞愧的。如果 Rails 也像它所做的“改革，简化”一样，花同等的时间分析/改进性能，很难相信每次版本升级我们会承受 40%-100%的性能退化。或许与 New Relic 的合作可以帮助 Rails 团队看到，对那些基于他们的平台而创建的实际应用来说，他们的决策具有何种现实的影响。

如果其他人的经历与我们相似，那么可以说这是许多人感受到的许多的痛苦。

我承认我有点不愿意写这篇文章，因为作为一个平台 Rails 给了我们太多，而且目前我们的业务太小，还不至于能直接牵涉到 Rails 的性能改进。但是我们将继续发表任何我们发现的、明显的优化策略，发表到这个博客或者别的什么地方。

不过我最关心的，同时也是我发表此文的原因是，如果 Rails 继续以这样的幅度降低速度，我不确定是否在 4.x 或 5.x 系列版本中，是否会有“没有返回的时间点”，在那时它将慢到我们无法再做任何升级。每次我们追随的新版本都向着那种可能性迈进了一步，即使我们购买了史上最快的服务器，并且给编译器实现了史上最耗心血的优化。

还有人做过由中等规模到大规模webapp的Rails 2 -> 3 -> 4这样的升级吗？我非常渴望听到你的经验。当 Google 搜索“Rails 性能”时，只有很少的一点结果，这总是使我想知道其他开发者升级经验的更多细节。

(1)就像在兼容的 web 服务器上使用动态流一样，在某些情况下，新的缓存模型也可以提升性能。由于这篇文章的目的我是着眼于“性能”，而性能属于在服务器上运行的动态 web 应用程序范畴，这意味着（我们所关注的是）诸如解释请求，与数据库交互，以及呈现响应。

Rails 的详细介绍：请点这里

Rails 的下载地址：请点这里

原文

http://www.linuxidc.com/Linux/2013-12/93365.htm?utm_source=Tuicool_Weekly

《基于 Oracle 的 SQL 优化》作者谈 SQL 优化的重要性

在刚刚结束的 Oracle 技术嘉年华上，Oracle ACE、中航信资深 DBA 崔华发布了个人新书《基于 Oracle 的 SQL 优化》。InfoQ 对他进行了简短的采访，谈了谈这本书的写作初衷以及对 Oracle 的展望。

InfoQ：为什么会写这样一本关于 SQL 优化的书?

随着我这几年在国内 Oracle 圈子里知名度的提高，使得我有机会接触到各种各样有问题的系统，在做了大量的实际诊断和优化工作后我发现，绝大多数的系统的性能问题都是由于当初开发人员不懂 Oracle 数据库、不懂如何在 Oracle 数据库里写出高质量的 SQL 所导致。这样的系统的性能问题，单靠高水准的 Oracle DBA 来调整是非常痛苦的一件事情，很多时候是事倍功半。

为了能从根本上解决这种问题，扭转这种局面，我萌生了撰写一本专门的、系统的、从本质上阐述如何在 Oracle 数据库里写出高质量的 SQL 的一本书，这就是《基于 Oracle 的 SQL 优化》这本书的由来。我希望通过这本书教会所有阅读过本书的开发人员如何在 Oracle 数据库里写出高质量的 SQL 以及如何在 Oracle 数据库里对有性能问题的 SQL 做诊断和调整，这样就能从源头上保证由这些开发人员所开发的系统在 SQL 上是没有性能问题的，随之而来的那些由于 SQL 的撰写不当而导致的各种性能问题也就不复存在了，如果真能做到这一点，

我个人认为会是一件功德无量的事情。

InfoQ：SQL 优化对应用系统的意义是什么？

数据库性能优化是系统生命周期中不可缺少的一环，我们大多数的数据库都或多或少遇到一些性能问题。特别是随着业务的发展、数据量的增加、系统用户数的增多、以及系统之间越来越复杂的接口，都会导致系统的性能越来越恶化。怎么样去优化一套数据库，我们需要掌握很多的知识，包括存储子系统、操作系统、数据库甚至还有业务逻辑。很多人面对性能问题不知如何下手。

好在，数据库的性能问题至少有 80%以上都是由于 SQL 性能较差所导致，同时随着硬件性能的提升和硬件价格越来越便宜，硬件所带来的系统瓶颈相对越来越小，所以在大多数情况下，我们优化数据库也就是优化 SQL 语句。这意味着只要我们做好了 SQL 优化，我们在大多数情况下就能保证应用系统在数据库端是没有明显性能问题的。

InfoQ：面对 NoSQL 的冲击，传统关系型数据库的定位是什么？

我个人觉得 NoSQL、大数据相关技术并不会对像 Oracle、DB2 这样的传统关系型数据库厂商造成大的冲击，更不用谈取代了——因为它们各自的适用场景并不相同。

事实上，像 Oracle、DB2 这样的传统关系型数据库厂商并不排斥 NoSQL 和大数据相关技术，比如我看到 Oracle 应对 NOSQL 和大数据技术的措施为：

提供对 NoSQL 的支持功能（如基于 Berkeley DB 的 Oracle 自己的 NoSQL）

提供了专门针对大数据的软硬一体的解决方案（如 Big Data Appliance 大数据一体机）

我个人认为像 Oracle、DB2 这样的传统关系型数据库厂商未来的定位会是拥抱像 NoSQL 和大数据这样的新技术并朝着云数据库的方向发展——未来会出现大规模公有云和私有云的盛行，到时候用户用数据库就好像现在用水电一样方便，是真正的 DBaaS（Database as a service)。

《基于 Oracle 的 SQL 优化》适合使用 Oracle 数据库的开发人员、Oracle DBA 和其他对 Oracle 数据库感兴趣的读者，也可以作为各院校相关专业的教学辅导和参考用书。该书目前已经在 China-Pub 和亚马逊预售，详细内容可参看相关页面。

原文

http://www.infoq.com/cn/news/2013/11/importance-of-sql-optimization?utm_source=Tuicool_Weekly

图灵访谈 ：世界级 Oracle 专家 Jonathan Lewis：我很为 DBA 们的未来担心

jonathan Lewis 世界级 Oracle 资深专家，有 20 多年 Oracle 关系数据库管理经验。主要从事自由咨询顾问工作，其 Oracle 数据库引擎方面的培训课程和研讨会世界闻名。Jonathan 曾是 UKOUG（UK Oracle User Group）的负

责人，他著有多本 Oracle 方面的畅销书，并维护自己的 Oracle 技术博客。

图灵社区： Oracle 的自动化程度变得越来越高，很多需要人为介入的优化手段也变得越来越简单。在这种情况下，DBA 怎么才能跟上发展、更好地提高数据库的效率?

JL： 要想跟上 Oracle 的发展是十分困难的。我之所以无法完成我想要写的书，就是因为我总是跟不上（笑）。虽然我可以解决很多实际问题，就算面对全新的难题，我也可以通过了解 Oracle 的表现，结合它的工作原理，很快诊断出问题。但是我却无法反向推断，如果你告诉我某种情况，我可能无法说出 Oracle 会做什么。我现在已经很难再进行预测了，但是我可以根据问题找原因。跟上发展的意思我认为应该是预前判断 Oracle 的工作状态，要做到这些，我认为要有足够的经验。

阅读手册是十分重要的事。我认为市面上还有两三本书，也可以帮助你掌握必要的信息。Christian Antognini 有一本书叫做《Oracle 性能诊断艺术》，在这本书中，他讲述了很多 Oracle 运作的知识，以及如何利用工具找到问题，同时也提出了如何解决问题的建议，这样的好书会教你 Oracle 怎么工作，然后会举一些例子告诉你 Oracle 能完成什么。如果你仔细读过手册，也读过这些书，你就知道如何构造模型，如何用特殊方式建立两个表，如何选择查询，如何选择 Oracle 的特性……你可以亲手实践手册或者从《Oracle 性能诊断艺术》里学来的东西，然后自己创造出一个可以解决自己实际问题的办法。通过亲力亲为的实践这些知识，你就会知道 Oracle 可以做什么，什么又是错的了。我认为实践

是最好的学习方法。

但是很多人没有时间来做这些事。你的公司可能会说：9 点到岗，5 点之前不许下班。这样你就没有空余时间，一直都忙着做别人安排好的事情。每天重复做同样的事，一做就是 5 年，于是你没有时间跳出来，尝试一些新方法。

有两个办法可以帮你获取更多的经验。第一个就是像今天这样的大会是很好的机会。你可以离开你的工作，听听别人分享的经验。你会得到很多新的启发，也会找到解决问题的不同方法。得到这样的启发是大有裨益的。

另外一点就是，我作为一个咨询师，我工作的方式会让我每年大概接触到 20 家公司，我就会知道 20 种运作方式。大多数人只能有机会看到 Oracle 如何工作的冰山一角，但是我几乎有机会看到全貌。所以关于在 Oracle 上可以做什么，我可能更有经验。常规公司的 DBA 经常会说：我们公司在过去 5 年中就是这么做的，我们有其他的办法吗？DBA 们应该经常安排自己重头思考一下，忘掉自己惯常的做法和思路，重新读手册，看看能不能想到一些新的办法。对于我来说，如果我看到你们在做的工作，我可能会想出 5 种不同的方式来达到同样的目的，因为我见过这些方式。对于 Oracle 来说，完成任务的方式多种多样，而普通 DBA 只能看到他每天接触的方式。所以，要成为出类拔萃的 DBA，你就必须去了解和尝试不同的做法。这些学习会花费你的时间，但是这也是必须要做的事。

图灵社区：你作为一位独立 Oracle 性能专家是如何工作的？在接到一个新案子之后你询问的第一个问题是什么？

JL：我还是挺出名的，所以经常会接到电话或者是邮件说：我们这出问题了，你

下个礼拜哪天有时间能来一趟吗？我大概每周工作 3 天时间，我还要留出一天做一些很重要的试验，来了解 Oracle，了解更好的运作方式，剩下一天我会从事写作，无论是书还是博客。所以接到这样的电话，我通常都是有时间的，也许是下周，也许是下下周。

一般我在接到工作的时候，他们都会告诉我，哪里出了问题，需要我来做什么。在整个解决问题的过程中，我第一步需要了解的永远都是：你们想要这个系统做什么？出现问题时的状况是什么样的？并不是Oracle在干什么，机器在干什么，而是用户在做什么。然后他们可能就会说，银行系统处理了大批量的境外交易，每个月我们在做账目核对的时候，就会出现性能问题。这就会告诉我他们平常的进程是什么，当在某天引入特殊进程的时候就会出现严重的性能问题，这两个进程之间哪里发生了冲突，它们的运行机制是什么。经常会有这样的情况，在得知业务在做什么，然后转入 Oracle 问题的时候，我就可以预测出数据库服务器上 Oracle 出现的是什么类型的问题。一旦我得到整体的情况，就可以追踪到具体的代码了，我开始检查数据库结构，询问为什么要建立这个索引，想通过它做什么。然后重新运行，如果不行就导出性能数据，找出 AWR 报告、历史信息存储，以及其他 Oracle 能提供的信息，由此得出下一条线索。

图灵社区：我知道您满世界解决 Oracle 难题，现在您已经去过超过 50 个国家了。你有没有在这过程中遇到过很棘手很难解决的问题?

JL：中国应该是第 52 个国家了（笑）。确实有这样的情况，很难判断问题究竟出在哪里。有的时候问题其实是一个 Oracle bug，但是很难证明。有时候会遇到关于并发行为很稀有的问题，两件事必须严格在同一时间发生才能出现这个问

题。于是我就只能说有可能是什么出了错，于是解决方法很有可能就是需要绕过这个问题。有那么几次，我最终也无法说出是哪里出了问题，虽然我知道在什么条件会产生这样的问题，但是我无法给出足够的信息来从根本上解决这个问题，甚至都没办法提供足够信息找到 Oracle 向他们寻求产品改进。我们能做的只能是建立历史记录，证明问题确实发生了。还有几次，在我找出问题后，其他人在我之前找到了解决方法。在我从业的 25 年中，有一两次这样的情况。

图灵社区：你有没有打算建一个团队来从事现在的咨询业务？

JL：大家总在问我这个问题。他们总是说：我可以为你的公司工作吗？事实上，我没有公司。我觉得有了公司之后就要为这个公司负责，这样就没法一直做自己喜欢的工作了。而这两点对我来说都没有什么吸引力。我不想为找到合适的工作者而发愁，我无法给出这就是他们需要用来养家糊口的工作的保障。所以，可以说我从来都没有考虑过成立一个公司。

图灵社区：当你的孩子还小的时候，你不仅要写书，还要忙着满世界做咨询服务，你是如何分配你的时间的？

JL：在 1999 年，当我的孩子还很小的时候，我开始写我的第一本书，当时还没有人像我一样做 Oracle 的咨询工作，大家当时都忙着重新编写程序，所以我觉得很有必要写一本书。我下决心花上半年的时间写这本书，暂时放下平常的工作。我告诉我的家人：我要写书了！第二天，我的女儿对我说：“你还在写那本书吗？”她以为我花了一整天写书，第二天肯定应该写完了！

在我的孩子很小的时候，大概是 12 岁之前，我会尽量少出门做咨询，可以说我

基本没怎么出远门。在他们大一些之后，我重新开始我的咨询工作，但是我会保证周末一定在家。我不会在一周内工作超过 5 天。今天大概是我 25 年中第三次，我周末没有和家人在一起。

图灵社区：这可能是你作为独立专家的福利之一吧。

JL：是的，我已经把很多人引诱到加入这个行列了（笑）！我跟他们说：如果你们周末工作的话，就要花上两倍的时间，如果你们星期天工作的话，就要用上三倍的时间。

图灵社区：想以数据库作为事业的人通常面临两个选择，一个是作为 DBA，另一个是作为开发者。你对面对这两个选择犹豫不决的人有什么建议吗?

JL：对我来说最大的问题就是我所看到的 Oracle 应用，它们没有体现出大家所说的 DBA 和开发者之间足够的合作性。有些人接受具体任务，他们写的代码需要和 Oracle 数据库交流，还有人管理 Oracle 数据库，他们整天看数据库的 SQL。有时候他们会说：这人会不会写 SQL 啊？这写的是什么呀？我认为不应该有这样的分界线。如果你的应用背后有 Oracle 数据库，你应该知道数据库是如何工作的，这样开发者和 DBA 之间的对话就会更有效。

很多 DBA 抱怨，现在的数据库很大程度上就是按按钮。如果你是一位 DBA，你可能这辈子都在做支持，没有机会接触像我所做的这些有趣的东西。如果你想精通如今的 Oracle，并且享受你在 Oracle 上的工作，我认为你应该把自己放在两种工作的中间。这样你就会了解数据如何工作，就会享受和数据库交互的过程，也会喜欢和业务应用交流的过程，你会知道如何构建自己的代码和数据库更好的

交流。所以，“DBA 开发者”也许才是最合适的位置，并不仅仅作为一个程序员，而是一个和数据库有亲密接触的程序员，享受数据库所能提供的帮助。当然，这可能需要找到一家能够提供这样位置的公司。

图灵社区：对于内核和功能，你认为关注哪个更重要?

JL：我认为每个人都应该熟悉 undo 和 redo 的运行方法，每个人也应该了解一些类似库缓存如何工作的知识，因为他们应该知道，在完成一个很简单的任务时，后台都发生了些什么。也许你不需要对所有事情都知道得一清二楚，但是你需要知道什么时候需要干什么，需要对整体有一个把握。一旦你对所有事都有了大概的了解，我认为你不需要再有更深的考虑了。大多数人需要知道的就是索引表怎么工作，群怎么工作，压缩索引有什么好处。当你改变数据的时候，当你需要优化的时候，你就知道如何避免信息阻塞。基于此，你就可以去研究一下物理设计，这就是你需要掌握知识的上线了。

我对 OTN 数据库论坛上的一件事感到很有趣，几乎每周，论坛上都会有一个问题“如果我把这行上移到这里，而不提交，在这之后我们运行了很久，但是突然间数据库崩溃了，Oracle 会如何处理这件事？”我对这类问题出现的频率感到惊讶，因为从很多方面来说，需要了解这些事的人非常少，你不需要知道如何解决和为什么这么解决，只需要知道它就是这样就够了。多掌握一些知识当然好，但是如果这需要耗费你大量的时间和精力的话就不划算了。人们总是不厌其烦的问我这些问题，这其实也是我写这本书的一个原因，你们来看书吧，别问了。

图灵社区：如果您的子女对计算机科学感兴趣，你会让他们选什么具体的方向呢?

JL：我的孩子对计算机科学一点都不感兴趣，甚至对科学都没什么兴趣。他们主修的都是艺术。如果要我给其他的人建议的话，我其实没法给出具体的建议。我对自己从事的工作很喜欢，我也鼓励别人做自己真正感兴趣的工作。如果你能通过这件事赚到钱，那就去试试吧。我不会和任何人说，你去做 Oracle。我知道有些人他们并不对 Oracle 感兴趣，只是因为他们做的不赖，做其他的事又做不来，所以才从事这份工作。刚才这个问题的实质其实是预测这个市场：5 年之后的就业市场会是什么样呢?

如果有人要根据未来就业市场的走向来定夺自己的未来。那我就大胆猜测一下。我认为对于 Oracle 顶级专家来说，会有一个很小的市场。5 年后，可能很大部分的普通 Oracle 工种都会被挪到云端，只会有不多的用可插拔数据库的支持，来维持很少数量的数据库。而大部分小型企业，只要租赁 Oracle 的云端数据库就可以了。他们不需要自己买硬件和软件，也不需要有自己的 DBA，只要为自己的应用租赁一块可运行的数据库就可以了。如果你想在 5 年后成为 Oracle DBA，你要么就是为很大的公司工作，他们自己为了保护数据的隐密性，而把数据留在自己这里。要么就是为很大的机器工作，500 台机器上运行着上百个数据库。小公司里的多数 DBA，就不会存在了。除了 Oracle，也会有其他的数据库技术也向这个方向发展。

事实上，需要维持大数据库的大公司数量并不很大。无论你的技术是什么，DBA 都不会很多。我将在 5 年内退休，对此我长出一口气。很高兴我不是个初出茅庐的专家，因为我很担心在 5 年后很多 DBA 的命运，他们找工作也许会很困难。除非他们真的出类拔萃，而且大家也认可他的能力。

图灵社区：你会考虑开发某种工具让代码可以跨越各种版本的限制吗?

JL：绝对不会。我从来没有想过要把我做的东西变成某种产品。因为如果要有产品的话，那个产品就一定要尽善尽美，而且要完美适配所有版本的 Oracle。这就意味着我要花很多时间对已经存在的东西改来改去。这对任何人来说，都是得不偿失的。我甚至都不会公布我的 SQL 代码，在我的博客上只有很少几个，我会说：这是一个很小的例子，这是 2003 年在某个版本的 Oracle 上运行的代码，它的作用是告诉你这个东西大概是怎么完成的。我绝不会说：这是一个在所有 Oracle 上都能运行的程序。看着我现在用的某些代码，我认为我可以把它们改得效率更高，这个高效版本可以在 10 上运行，也可以在 11 上运行，我十年前写的代码，现在仍然可以在 9，10，11 上用，对于我的要求来说完全适用。但是如果要我做一个产品来解释，我就需要它在所有产品上适用，而且还要回过头修改以前的代码。人们不会为不产生新的结果的事情付钱。所以从经济学上来说也是不合算的。

市场上有些工具，我不会说具体名字，当你仔细观察它们运行的 SQL，你就会发现他们的效率很低，这不是做事的正确方法。但你会认识到，在十年前，没有改变的需求的时候，它们仍然能够运行。在很多年以前，当 Oracle 9 刚问世的时候，有一个公司有个这样的标准工具，我在一个大会上看到了，是一个很小的展台。他们有一个产品在 Oracle 8 上小有名气，当时展台上挂着一个巨大的条幅，上面写：“Oracle 9 的保证”。我就去问他们一些细节：“你们说 Oracle 9 的保证是什么意思?你们能做这些那些吗？”最终他们终于承认，这句话的意思其实是，代码在 Oracle 9 上仍然可以运行。但是这样的产品并不是为 Oracle 9

设计的，而是为 Oracle 8 设计的，只是可以在 9 上运行罢了。可以说如果我是这个产品的顾客的话，我听到这些话可不会高兴的。但是商业公司有自己的考虑，也就是这种考虑上的差别，使得我做梦也不想把我做的东西变成产品。

图灵社区：《Oracle 核心技术》和《基于成本的 Oracle 优化法则》是两本很受欢迎的技术书。在《基于成本的 Oracle 优化法则》中你说过，会有一系列关于此主题的书，《Oracle 核心技术》是这个系列的第二本，这个系列的其他书也在准备中吗?

JL：事实上，《Oracle 核心技术》并不是这个系列的第二本书。当我写完《基于成本》，我决定不再继续写那些复杂、耗费时间，但又不是十分必要的主题，比如分布查询和平行查询。因为这些很让人费神的东西的需求并不很大。但我确实有计划继续写书，主题是我所掌握的其他信息。这个决定的根源是 Oracle 的变化和升级太快了，就算是有我这样背景的人，也无法跟上 Oracle 发布的步伐，无法预测在所有情况下 Oracle 的反应。虽然我一直尝试紧跟发展，写一本关于优化，内容丰厚的书，但是却一直无法开始。

《Oracle 核心技术》其实是倒退了一步，它跟随的是我在 2000 年写的一本书《Practical Oracle 8i》，介绍了 Oracle 工作的核心机制。《Oracle 核心技术》扩展讲述的就是这本书一、二章的内容。我要小声透露一个消息，我可能会写一本关于物理结构、索引，以及表方面的书，但是这件事还没有确定。这本书是要把《Oracle 核心技术》中尚未展开的部分写出来，教你在写代码之前，通过如何选择不同的物理结构，让数据库变得更有效率。如果可能的话，这将是一本很有意思的书，但是让我完成另一本关于 Oracle 的书希望渺渺。《Practical

Oracle 8i》是一本好书，现在仍有人在买，如果我真的能完成下一本书，内容会和这本书一、二章之后的内容相关。

原文 http://www.ituring.com.cn/article/62995?utm_source=Tuicool_Weekly

映射的存储模型 – 面向列的存储和行存储

1 个月没跟大家见面了，这个月真的是很累，做了很多事，天天加班，不过结还都不错，以后调休吧 ~

双 11 不知道各位感觉如何？是不是觉得很平稳呢？ 在这次双 11 里面，中间件团队配合业务团队使用了一种全新的全链路压测方式进行了线上性能验证，提前就预演了所有可能的压力，因此这次双 11 是我五年双 11 支持过程中最淡定的一次 ~

然后，又使用了阿里中间件的全部技术支持了某大型公司的企业的业务逻辑重构。让他们的系统业务架构有了一次全新的变化，让他们的系统具备了自由扩展的能力，从此不再担心系统无法支持用户增长了。顺便也就验证了我们解决问题的整体思路是一个可复制的思路，将会在未来有更广阔的发展空间 – 玩法变了啊，各位感受到了么？

好，话题回到模型上，今天我们开启一个新的主题，就是映射的存储模型。换句话说，在了解了 k-v 的基本机制之后，下一步我们研究一下我们的数据应该如何使用映射。

回顾一下映射，映射的本质就是一个 map，给出 map 的 key， map 就返回 key 所对应的 value。我们也介绍过，在计算机里的大部分事情其实都可以被表示为一个 map，比如，给定扇区号，返回给定扇区的数据等等。

我们已经知道了 Map，而在这里我们最需要解决的问题就是，给定一组数据，应该如何存放到我们的映射里呢？其实存放的方法非常多样，而核心的问题是用户的实际需求。

这里我们需要举个例子，假定目前我们需要存储的数据是一组数据。

bizOrderID	sellerID	buyerId	content
0	0	4	‘a’
1	0	3	‘c’
2	0	2	‘l’
3	0	1	‘d’

4	3	4	‘b’
5	2	4	‘f’

针对这样一组数据，我们其实是有很多种不同的存储方式的。

第一种，以 pk 作为 key，以其他数据作为 value

Map	
Key	Value
{bizOrderId:0}	{sellerID:0,buyerId:4,content:’a’}
{bizOrderId:1}	{sellerID:0,buyerId:3,content:’c’}
{bizOrderId:2}	{sellerID:0,buyerId:2,content:’i’}
{bizOrderId:3}	{sellerID:0,buyerId:1,content:’d’}
{bizOrderId:4}	{sellerID:3,buyerId:4,content:’b’}
{bizOrderId:5}	{sellerID:2,buyerId:4,content:’f’}

可以看出，在这种存储的实现中，每一行的数据都冗余了列的名字和该列所对应的类型信息。

这种方式有个专用的名词，就叫”面向列的存储“，当然，虽然出现了“列”字，但各位可千万不要望文生义，这面向列的存储，跟我们后面要看到的列存基本不

沾边。。

这种存储方式，最大的好处就是每个 value 里面的数据可以完全自己定义，目前主流的实现是使用 json 来存储 value 数据，这样，如果业务要求增加一个列，那就在 json 拼装的时候额外增加一个列就行了。而如果业务需要减少一个列，也可以直接在代码里拼装，减少了运维的成本。

不过这样做也不是完全没有代价，额外的冗余数据就意味着额外的空间消耗，所以目前通用的优化方案，利用了列的个数本身是比较有限的，这个特性，于是利用一个新的 map_bit. Key 为列的名字，value 为一个 bit . 从而减少冗余数据带来的过多空间消耗。如下：

Map_bit	
Key	Value(bit)
sellerID	‘a’
bizOrderId	‘b’
buyerId	‘c’
content	‘d’

Map	
Key	Value
{b:0}	{a:0,c:4,d:’a’}
{b:1}	{a:0,c:3,d:’c’}

{b:2}	{a:0,c:2,d:’i’}
{b:3}	{a:0,c:1,d:’d’}
{b:4}	{a:3,c:4,d:’b’}
{b:5}	{a:2,c:4,d:’f’}

有了这么个结构，面向列的存储里面数据冗余的大问题得到了部分的解决，然而这种模式还有优化的空间。在大部分情况下，我们的业务模型基本上是稳定的，不会过于频繁的发生变化。如果我们能够有这样的假定，那么我们就可以将列所对应的位置固定下来，用一个叫 map_schema 的表格来存储，不就不需要冗余上面的那些’a’,’b’,’c’,’d’等 bit 信息了。可以更多地节省存储空间啊。

map_schema	
Key	Value
sellerID	表格内的位置:1
bizOrderId	表格内的位置:0
buyerId	表格内的位置:2
content	表格内的位置:3

Map			
0	0	4	‘a’

1	0	3	‘c’
2	0	2	‘l’
3	0	1	‘d’
4	3	4	‘b’
5	2	4	‘f’

这种方式就是我们最常见的”行存“了，这种方式的比较于面向列的存储，最主要的优势就是空间消耗更少，而且申请空间的效率更高，因为用户在开始的时候就已经指定了所有数据所需要的数据类型（空间消耗），因此消耗的空间是相对比较固定的。

而对于面向列的存储而言，由于不能提前假定每一行数据的大小，所以只能使用变长数据存储格式来存储数据。 因此也会有更多的空间浪费，并且提高申请存储空间的代价。

无论是面向列的存储，还是行存，他们的写入都可以只通过一次 map.put(key,value)就可以完成。所以写入的效率比较高，如果按照 pk 这一列做查询，速度也会非常快，因为也只需要一个 map.get(key)就可以取出需要的数据，速度自然也就是很快的。

原文 http://blog.sina.com.cn/s/blog_693f08470101omuq.html

使用 MySQL 自身复制来恢复 binlog

如果需要恢复的二进制日志较多，较复杂，强烈建议使用 MySQL 自身复制来恢复 binlog，而不要使用 mysqlbinlog。

在 MySQL 手册中一直是推荐使用 mysqlbinlog 工具来实现指定时间点的数据恢复，事实上，这是一个经常"让人郁闷"的办法。更好的办法是，使用 MySQL 内部复制线程中的 SQL Thread 来做恢复。

这个 idea 来自 Lazydba 同学；在 Google 稍作搜索，在 Xaprb 上 Baron Schwartz 也很早提到了使用类似的方法来恢复 binlog，在那篇讨论中，还可以看到 Jeremy Cole 也提到：使用 MySQL 手册中推荐的方法是困难重重的，而且 mysqlbinlog 这个办法从逻辑上来说也是一个错误--因为这样 MySQL 不得不在两个不同的地方实现一套相同的逻辑，最终难免会出错。使用 mysqlbinlog 来恢复，你可能会需要以下“让人郁闷”的问题：

(*) Max_allowed_packet 问题

(*) 恼人的 Blob/Binary/text 字段问题

(*) 特殊字符的转义问题

(*) 没有"断点恢复"：执行出错后，没有足够的报错，也很难从失败的地方继续恢复

1. 如何操作

本文不打算写一个 step by step 的文档，只介绍主要的思路和粗略的操作步骤。

1.1 将 binlog 作为 relay log 来执行

优点：实施简单；缺点：需要关闭一次数据库(不确定不关闭数据库行不行)；

思路：直接将要恢复的 binlog 拷贝到 relay log 目录，并修改 slave-info 相关的文件，让 MySQL 把 binlog 当做 relay log 来执行

简单的操作步骤：

* 关闭当前实例
* 将 binlog 拷贝到对应的 relay log 目录(datadir 或者 relay-log 参数指定的目录)
* 打开 relay-log-info-file 参数指定的 relay-log.info 文件(默认是 datadir 目录下的 relay-log.info 文件)，修改文件前面两行。

这两行的意义分别是：当前执行的 relay log 文件；当前执行到 relay log 文件的位置(position)

* 打开 relay-log-index 文件(由参数--relay-log-index，默认是数据目录下的 host_name-relay-bin.index)将需要恢复的 binlog 文件全路径列表存在该文件中
* 启动数据库，并 start slave io_thread

1.2 从专门构建的 binlog server 上拉 binlog

这个方法，无需启动数据库，但是需要重新启动一个全新的实例，将 binlog 拷贝到该实例中，这里称这个实例为 binlog server。然后把需要恢复的实例复制指向这个 binlog server。这里需要做的是，将日志拷贝到 binlog server 对应目录下，并修改对应的 master-info 文件，使得备库能够 dump 到这些 binlog 文件。

2. 其他需要注意的事项

* 配置文件中建议加上 skip-slave-start，以免在不需要时候 slave 线程自己开始执行了

* start slave 的时候，可以通过 start slave until 的方式，控制 slave 执行到的位点

* slave 执行的其实位点，则通过 relay-log.info 或者 change master to 来指定

Good Luck.

原文

http://www.orczhou.com/index.php/2013/11/use-mysql-replication-to-recove-binlog/?utm_source=Tuicool_Weekly

自动摘要算法

当时 yahoo 以 3000 万美元的价格收购了 summly 的消息传出来之后，貌似大家都比变的对自动摘要产生了极大的兴趣，关于自导摘要 wiki 这里有很详细的介绍，一般自动摘要比较常用的一个是摘取文章中的关键词，另一个则是摘取文章中的关键的句子，在这里我主要是介绍用 textrank 算法来搞句子的摘取。

相对于 textrank，摘取关键句子还有一些比较简单的算法，比如这篇，我们可以把句子分别和整篇文章做比较，相似性最大的就是关键的句子。而 textrank 其实就是 pagerank 算法扩展到句子上，来的到一些全局的信息。textrank 的论文在这里。

首先我们看 pagerank 算法，就是 google 用的那个网页排序的 pagerank 算法，首先网页之间都是有超链接链接的，如果某个网站 A 有指向 B 的超链接，说明 A 网站认为 B 网站是有价值的，于是相应的我们可以给 B 来提升权重，但是就像现实中，一般人的意见和专家的意见的权重是不一样的，所以如果网站 A 的权重比较高，那么就可以贡献更多的权重给 B，反之则贡献更少的权重，然后算法经过一轮轮的迭代，所有结点的权重会收敛，就得到了最终的权重了。然后我们有 pagerank 的计算公式，其中 d 是阻尼系数。

$$score(V_i)=(1-d)+d*\sum{\frac{1}{|out(V_j)|}*score(V_j)}$$

换到 textrank，对于 pagerank 而言，边是没有权值的，而 textrank 的边是有权值的，表示两个句子的相似性，这个权值可以用 Jaccard similarity coefficient 就是交集数目除以并集数目，或者 cosine 的余弦夹角，或者是 bm25 一类的算法。在边有了权值之后，textrank 的公式变为这个样子。

$$score(V_i)=(1-d)+d*\sum{\frac{weight(j,i)}{\sum{weight(j,k)}}*score(V_j)}$$

论文里显示带了权重之后收敛会慢一些，在经过若干轮的迭代之后，我们就可以得到每个句子的权重了，然后我们取出权重最大的几个就认为是文章的关键句子了。自己实现了一份以 bm25 为相似性的 textrank 算法，代码在 github 上，具体可以看代码，test.py 展示了摘要的提取。同时这个项目实现了一些常用的操作，比如繁体转简体，中文的分词和词性标注等，欢迎试用提 pull requests 啊！

原文 http://www.isnowfy.com/automatic-summarization/?utm_source=Tuicool_Weekly

关于 MySQL count(distinct) 逻辑的另一个 bug

上一篇博文(链接)介绍了 count distinct 的一个 bug。解决完以后发现客户的 SQL 语句仍然返回错误结果(0)，再查原因，发现了另外一个 bug。也就是说，这个 SQL 语句触发了两个 bug -_-

这里只说第二个，将问题简化后复现如下，影响已知的所有版本 。

```
drop table if exists tb;
set tmp_table_size=1024;
create table tb(id int auto_increment primary key, v varchar(32))engine=myisam charset=gbk;
insert into tb(v) values("aaa");
insert into tb(v) (select v from tb);
insert into tb(v) (select v from tb);
insert into tb(v) (select v from tb);
insert into tb(v) (select v from tb);
insert into tb(v) (select v from tb);
insert into tb(v) (select v from tb);

update tb set v=concat(v, id);
select count(distinct case when id<=64 then id end) from tb;
```

返回 64，正确

select count(distinct case when id<=63 then id end) from tb;

返回 0

上述中 update 语句的目的是将所有的 v 值设为各不相同。

与上个 bug 类似，5.5+的版本直接复现；5.1 版本需要修改的是 max_heap_table_size 参数，而由于 max_heap_table_size 的最小值限制不能设置为 1024，需要的测试数据量大些，但原理类似。

原因分析

Count(distinct case when xxx then f end)的语义就是计算字段 f 的去重总数，计算流程细节参看前一篇。这里直接给出 tmp_table_size 不够大时的流程，便于说明此问题。

流程：

1、 构造一个 unique 集合 A1，将满足条件的结果插入 A1 中（计算了 case when 之后的值）

2、 插入 item 过程中若大小超过 tmp_table_size，则将 A1 暂时写到文件中，再构造集合 A2

3、 重复步骤 2 直到所有的 item 插入完成

因此若 item 很多则可能重复生成多个集合 A1～An。

4、 对 A1～An 作合并操作。由于只是每个集合 A 保证 unique，因此需要做类似归并排序的操作（实际上不需要排序，只是扫一遍）

5、 合并加和操作本来只需要去重和去掉 NULL 值即可，但为了复用代码，对于每个 item，重新计算了一次结果的合法性，也就是，再判断一次 case when 是否正确。

6、 不幸的是，计算结果合法性的这些 case when，其实是共同的一个：最后一行。

因此最后的结果是正确值还是 0，就取决于最后一行的 case when 的结果。

案例分析

以上面这个 case 为例。由于使用主键，最后的一行必然是 id=64 的那一行。这样在合并的时候，若条件是 id<=64 这些值都被认为符合条件可以合并。而最后一个语句的情况，最后一行 id<=64 不成立

作为验证可以看一下这个 case

```
CREATE TABLE `tb2` ( `id` int(11) NOT NULL , `v` varchar(32) DEFAULT NULL ) ENGINE=MyISAM DEFAULT CHARSET=gbk;
insert into tb2 (select * from tb order by id desc);
select count(distinct case when id<=63 then id end) from tb2;
返回 63，正确
```

可以看到，其实 tb2 和 tb1 的数据内容是一样的，只是 tb2 没有索引且数据倒置插入，因此查询的最后一行的 id 是 1，满足 id<=63, 结果记入就正确了。

解决方法

调高 tmp_table_size 也是一种直接的方法，但是不治本，因为只要满

足条件的行数足够多，就会出现这个问题。

当然本质上这是一个 bug。

代码上，对于已经走到合并操作的这个逻辑，其实前面在构造各个集合 A1~An 的时候，已经验证过条件合法，其实在合并的时候，可以直接做去重操作即可。

http://dinglin.iteye.com/blog/1982176?utm_source=Tuicool_Weekly

有关于存储过程的一个笑话

我真的是上来讲一个笑话的，这是个传统笑话，需要慢慢讲。

我同事离职以后，统计平台的所有代码就被我接过来了。这个统计平台是用 MySQL 搭建的，5.1 版本，服务器是 DELL 的，有 16 个 CPU 和 32G 内存，700G 的 data 空间，说真的这个配置对于我这种喜欢 IOE 的人来说实在是看不下去。

接过这个系统后我发现有一个过程需要跑三个小时，从前 4 点开始跑，到早上 9 点前也就出来了，于是也就没有什么。可是后来因为生产服务器压力很大，ETL 的时间被推后了很多，导致我的过程只能 7 点开始跑，这样子，加上数据入库，这个系统的日报往往要 11 点甚至 12 点才能跑出来，领导问我为什么没有数据的时候我真的不知道该如何狡辩。于是我想到了把计算过程迁移到 Oracle 上。我也这么做了，效果很好，这个三小时过程 Oracle 只需要不到 1 分钟然后 spool 出来 load data 都比原来的过程快很多倍。于是我就安心的用起了 Oracle。

但是我今天工程师精神泛滥了，我想知道为什么同样是数据库，人家阿里能把性能调校的那么好，我们就要等那么久?

我跟踪了一下，最慢的是这样的一个 SQL：

```
select distinct b.name
from table_a a
inner join table_b b
  on a.id = b.id
group by b.name
```

其中 A 表 1 亿余数据，B 表 2 千万左右，数据量可以说已经很大了。a 的 ID 有一个索引，explain 了以后这个 SQL 也走了索引，两个表都走了。这就很奇怪了，于是继续看执行计划，extra 那里有个 temp table 还有一个 filesort。于是我开始想笑，这是谁啊，distinct 了还要 group by 一下 b.name？于是我检视了一下 b 的表结构，这个表上的 name 是有一个索引的。我于是懂了从前的程序员为什么要画蛇添足的加一个 group by。这在 Oracle 程序员看来是愚蠢的行为里，其实蕴含了一个 MySQL 的小技巧。我接触 MySQL 不长时间，但是通过读各种书籍发现 MySQL 的索引很有趣，尤其是聚集索引，整个 InnoDB 表就是一个索引。

扯远了，这个 SQL 里，就是因为这个 group by 让 b 表也走了索引，但是，当时的程序员应该是忘掉了去掉前面的 distinct，于是导致了悲剧的发生。

这个 SQL 完整版是这样子的：

```
insert into table_c(day_id, name, flag)
select '20131128', distinct b.name, 1
from table_a a
```

```
inner join table_b b
  on a.id = b.id
group by '20131128', b.name;
```

去掉了 distinct 以后这个 SQL 还是很慢，半天插入不了。table_c 有很多索引，这是个典型的 DW 表，很大。于是我建了一个中间表，只有一个字段 name，可是我发现插入的速度还是很慢，非常慢，于是我想起了几天前我看的《高性能 MySQL》，里面 169 页还是 179 页讲过页分裂的概念，提及了一个小技巧，就是每个表都应该有一个自增的主键，这样子插入速度会有一定的提升，于是我把 mid 表改了，有两个字段：id，name，其中 id 是自增的主键。这样一来，插入 mid 表只需要 5min 左右，而把 mid 表用简单的 select 语句插入 table_c 只需要 20s。是我想要的效率。

我学习 MySQL 只有短短的几个月时间，而且没有时间从最基础的开始学，都是需要什么知识就赶紧去看《高性能 MySQL》，这本书真不错，当做一本救急用的字典确实可以满足作为一个开发人员的需要。当然了，我的定位是开发型 DBA，这本书里讲管理入门的还比较少，我还是要好好学习。欢迎大家指正。

原文

http://blog.csdn.net/uncle_six/article/details/17031571?utm_source=Tuicool_Weekly

揭开程序员装 13 行为的面具

[核心提示] 程序员一直都是很善良的 IT 工种，勤勤恳恳不辞辛苦的工作，不过今天可不是为了夸程序员。来 818 程序员有哪些装 13 的行为，别说没有。

程序员一直都是很善良的 IT 工种，勤勤恳恳不辞辛苦的工作，不过今天的文章不是为了宣扬程序员的伟大。尽管在互联网的发展中，他们贡献了无数的代码，用自己的技术推进了互联网的进程。我们还是要扒一下程序员的装 13 行为，可能会有很多程序员看了本文会十分的愤慨，但考虑到你们很忙，没有时间黑公园网站，我也就不客气了。

员你还说没有装 13

写代码离不开各种编程工具，有众多工具供选择便有花样的喜好，对装 13 的程序员来说，是坚决要抵制 IDE 的，IDE 臃肿缓慢，一定是要用 vim 加编译器的组合。vim 和 emacs 就代表高端，用 IDE 就是 low 。我不否认很多大牛使用 vim，但也有不少写不出好的程序还要用 vim 装的，很多明明用起来很吃力，操作不熟练，好像非要用 vim 才能写出优秀代码一样。

用 vim 配合各种快捷键、扩展觉得顺手，加上 vimer 本身优越感，其他的编辑器一如 emacs 和 notepad++ 之流就是渣，深爱一种编辑器便唾弃其他的，即便是口上不说，在黑客马拉松上看到别人在用什么 UltraEdit 写代码，内心也会鄙夷一番，顿生自己很牛 X 的幻觉。

还有自认为 Mac 写出来的代码比 Windows 写出来的优秀，去咖啡馆看一看，拿 Mac 的就两种人，不疼不痒的文艺小青年和自以为是乔布斯的码农。

听说写代码很牛的工程师都用 Happy Hacking Keyboard，在好不容易挣了点钱之后，狠下心花了 2k 多买了个 HHKB 键盘，还必须得是无刻字版。看着清一色没有任何字母的键盘，一想到也用上了这么高端的键盘，好像技术能力也飙升了一样。技能不怎么样，装备是不能落下的。

鼠标的发明让用户使用电脑的门槛降低了很多，图形化的界面加上自由移动的鼠标点击方便了操作。在程序员眼里并不是这样的，鼠标对他们来说就是累赘，它是效率的杀手。一定要用各种指令，所有的操作都在键盘上狂敲，这样才能体现出技术水准。

对于某些崇尚开源文化的程序员来说，只有开源的软件才能让他们兴奋，不开源的软件都是受到异样的眼神看待。甚至在他们眼中，是不能理解为什么会有人写出这么烂的 Windows 。他们恨不得所有的软件都是开源的，这样就可以更多的复制那些优秀的代码，而那些自己写的程序则是不希望让外人看到源代码，大多是因为自己代码写的太烂。

不加班的程序员不是好程序员，他们经常以自己在深夜编程为荣，甚至宣称在深夜开发才有灵感。最好还要在半夜发条状态：每天看着星光回家感受特别充实之

类的。实际上的原因很简单：碌碌无为的白天引发的愧疚心。

某些自身的需求，比如在豆瓣租房小组里找房，一般人浏览一下小组内容，就可以获得自己需要的信息。程序员们一定要用高大上的方法，要写个脚本，抓一下数据，然后再根据自己的需求关键字检索一下，否则不足以凸显程序员的独特技能。

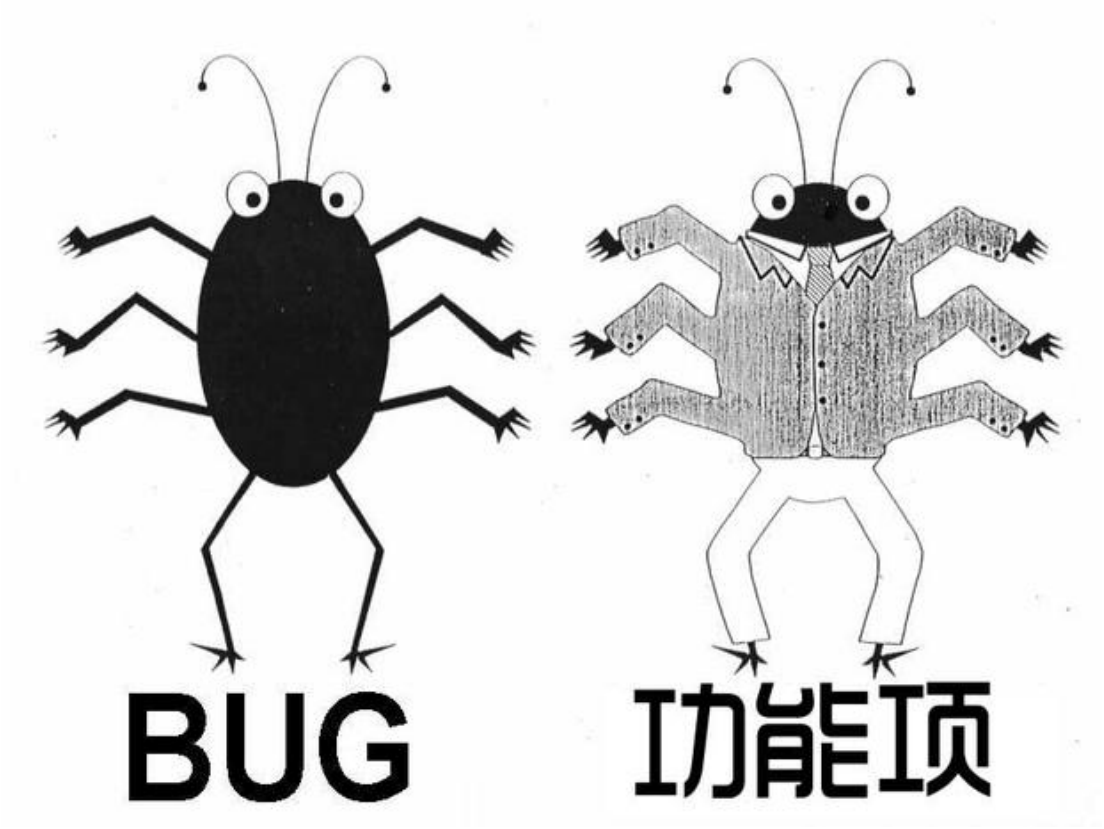
(via)

对于产品的升级，增加新的功能，程序员们会以工程难度大、很难实现为理由，抵触做一些改进，如优化之类的。已经做好的工作不想改动，那么背后真实的原因是什么呢？往往是之前写的代码太烂了，不愿意回首再读，设计之初没有考虑扩展性。他们甚至宁愿重新写，也不愿再改动代码。

在一些上司不太懂技术的公司里，一个项目分配下来之后，程序员会用各种专业的技术语言来跟上司沟通，用上司听不懂的重构、迭代等术语来“阻止”这个项目，实际上不就是为了给自己多争取点项目时间。

程序员的世界里，是恨不得所有的电脑都和他的一样。用最后一点耐心完成某个

项目的代码后，提心吊胆的在自己的电脑上运行没有问题了，提交上去之后，服务器怎么就跑不动了。在我这运行没有问题的程序，为什么在你这边就出现问题了，一定是你的电脑配置有问题。反复问清楚你的情况之后，确定这不是你的问题，才承认是自己程序的 bug。



（via）

本来能 10 行能解决的程序，一些程序员一定要把它拆开，一层一层的用设计模式去做，不断的面向对象的包装，包装的一层一层又一层，码出来 100 多行代码，这样他们才感到高兴。虽然不会太影响执行效果，但代码让人无法看，于是下面每一个读代码的程序员都会觉得上一个写代码的是傻 X。

就是蒙蔽自己的程序员

互联网技术的发展中，程序员的功劳毋庸置疑，他们用一行行的代码打造了很多产品，丰富了我们的各种体验。他们有着心怀用技术改变世界的梦想，但种种对技术的偏执也产生了很多装 13 的行为。上文列了这么多的症状，程序员这么一善良的物种未能幸免。我们来看一看程序员装 13 行为背后的动机。

在电脑发展初期，还没有图形界面的时候，一提到黑客、程序员，我们都会自行脑补他们面对着黑压压的屏幕，敲着各种字符。vim 和 emacs 纯文本的工作环境也让他们沉浸在写代码的快感中，久而久之 vim 就代表着老练，以至于后来有些程序员崇尚 vim ，以为用 vim 就代表技术能力强。使用通用编辑器并不是问题，但熟练编辑器的使用和技术能力强是两码事， 当你所常用项目中有某个 IDE 拥有十分吸引你的特色功能时，尝试使用它。何苦逼着自己很吃力的

用 vim ，跟自个过不去。

程序员喜欢用自己的技术来解决一些问题，这本不是装 13 的行为，可以说是值得鼓励的，很多伟大的互联网产品都是从最初一个不经意的意外尝试开始的。这并不是意味着遇到问题就首先想到用代码解决，花一大段时间来解决一些技术难度不大的问题，没有实质性的意义。非要用技术来解决这个问题，凸显自己好像技术很强的样子，就是装。

也不知从什么时候开始，加班成为程序员的习惯。在各种传奇的创业团队经历多少个日日夜夜开发的故事鼓舞下，以为半夜开发就能写出很牛 X 的代码，就能有更多的灵感。不断的这种事迹，让他们逐渐在潜意识中植入白天写不出好代码的概念，于是白天的工作效率也就很低下，而这种靠深夜开发获得成就感的行为，实在是可悲。

装 13 的本质就是不愿意承认事实，用一种假象来蒙蔽别人。对程序员来说，不愿意做一些改动，优化工作，不能理解为什么在自己电脑上运行没有问题在服务器上就跑不了，这种情况就好解释了。他们找出各种理由来搪塞，试图掩盖事实。真相大多是因为之前的代码写的太烂，可读性不高。而当自己与其他人合作的时候，看到别人写的代码，心里也会默默的鄙视一番，实际上你写的也好不哪去。

都知道了，你装 13 给谁看

程序员是需要不断的学习的，在还没有写出牛 X 的产品的时候，好好打怪升级。切合自己的实际需求，来选择合适的工具，技术不怎么样的非要硬逼着自己装出很牛 X 的感觉，也就骗骗自己，技能不够用再好的装备也发挥不出来水平，相

信玩过游戏的都有过经验。也别非遇到什么问题都想着写个程序来解决，对这样的程序员们，我有一个问题，你现在有女友吗?

一个简单的问题，好像必须复杂化才能凸显自己的能力似的，于是想方设法的用各种代码设计，码完了自己看着是开心了。等回头需要改动的时候，代码写的一团糟，自己都不忍心回首，就别找理由来忽悠别人了。

原文 http://www.geekpark.net/read/view/194170?utm_source=Tuicool_Weekly

6 家科技公司 CEO 分享第一份工作经验 :有除草的，有搬家的，有处理污水的...

无论一个人日后变得多么的成功，无论你的职业会把你带向何处，但是每个人都会有一个起点。下面是六个科技企业 CEO 们分享他们的第一份工作的经验，有的人是在污水处理厂工作，有些人在为搬家公司工作，具体如下：

Mark McLaughlin, Palo Alto Networks 公司 CEO：直升飞机驾驶 22 岁

Mark McLaughlin 带领 Palo Alto Networks 公司在 2012 年成功上市，现在价值 33 亿美元；他之前是 Verisign 公司的 CEO，2010 年时以 12 亿美元的价格

出售给了赛门铁克公司。

你的第一份工作是什么？

我有许多的打工经验。比如 12 岁时帮人打理草坪；15 岁时帮人换燃气。16 岁到 18 岁之间，干过两个工作麦当劳和杂货店。

你最奇怪的工作经历是什么？

我的职业生涯开始是直升机驾驶，当时我在美国空军服役，历时两年，之后我进入了法学院。(另外，在美国空军期间，Mark McLaughlin 获得了立功奖章)

Paula Long, DataGravity 公司 CEO：采烟叶

DataGravity 是一家存储技术公司，获得了 4200 万美元的投资。Long 之前和人共同创建了 EqualLogic 公司，在 2008 年以 14 亿美元的价格卖给了戴尔公司。

你的第一份工作是什么？

14 岁的时候，在康涅狄格州采烟叶，一共干了两个暑假。

你最奇怪的工作经历是什么？

我有过人工智能和机器人的工作经验，控制机械臂完成生产线上的工作。

Mike Gregoire, CA Technologies 公司 CEO：污水处理工作

CA Technologies 是一家 37 年历史的企业级软件公司，年收入 46 亿美元。在此之前，Gregoire 是 Taleo 公司的 CEO，在 2012 年以 19 亿美元的价格把它卖给甲骨文公司。

你的第一份工作是什么？

15 岁的时候，我给一家家具公司做送货业务，一年之后，那家公司倒闭了。

你最奇怪的工作经历是什么？

我曾在一家污水处理厂工作，非常脏也很辛苦，而且我的领导认为我干的不好，其实我是喜欢有创造性的工作。

Jayshree Ullal, Arista Networks 公司 CEO： 井下电缆检测

Arista Networks 公司计划 2014 年上市，估值 25 亿美元。在 Arista Networks 之前，Jayshree Ullal 负责思科的路由器业务，将这个部门的营收增长到 100 亿美元。

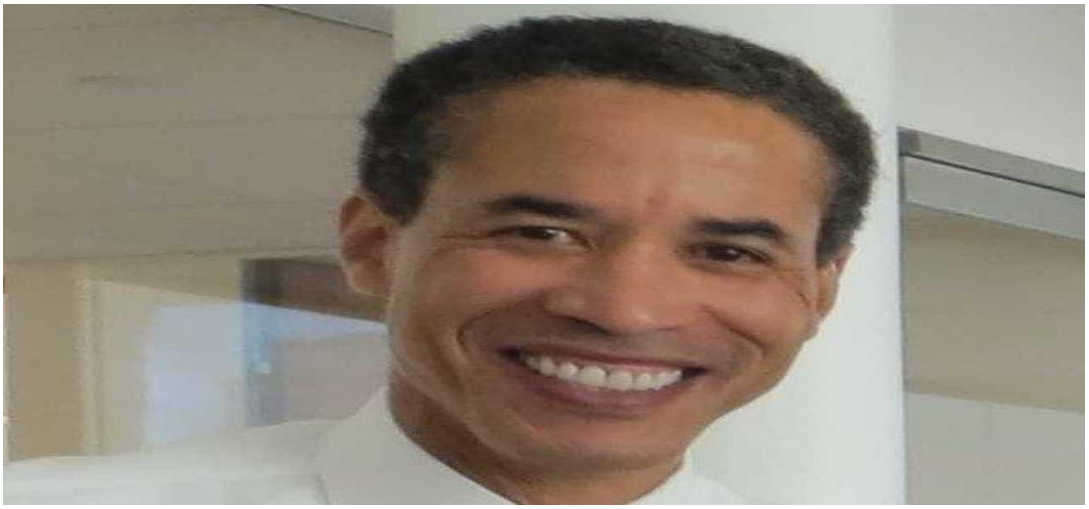
你的第一份工作是什么？

16 岁开始在 Woolworths 做收银员，干了三年。

你最奇怪的工作经历是什么？

我曾在 PG&E 公司做电子工程师的实习工作，负责测量井盖下电缆的电压和电流，需要下井工作的。这个经历让我体会到工程师需要灵活和自由的精神。

Charles Phillips, Infor 公司 CEO：搬着冰箱上二楼



Infor 公司是一家企业级软件公司，营收入在 30 亿美元左右，之前是甲骨文公司的第二号人物。

你的第一份工作是什么？

13 岁时我给人除草。我的父亲还要求我付汽油钱，实际他是想教我如何做生意。一段时间之后，我发现可以使用客户自己的割草机，这样就不用自己付油钱了。

你最奇怪的工作经历是什么？

我 16 岁的时候，给一个搬家公司工作，这个经历很辛苦。有时候需要搬着冰箱上二楼;还有的时候，在搬家的途中客户会改变主意等等。这段经历让我学会了如何通过电话交流，迅速的判断客户的情况，比如家里有几个孩子，几个房间等。17 岁的时候，我给一家银行的分支机构，写个应用程序，这是我在 IT 界的第一份工作。但是最让人激动的还是在海军陆战队的工作，那里的每个人都极度的认真，为了不让自己的队友受伤害。

Jim Whitehurst, Red Hat 公司 CEO：为一个股票经纪人开发一款程序

Red Hat 是 Linux 软件公司，去年的营收额为 13 亿美元，到 2016 年会增长到 30 亿美元。之前，Jim Whitehurst 是 Delta 航空公司的 COO.

你的第一份工作是什么？

1980 年代，电脑和编程还不是这么普及的时候，我的第一个工作是为一个股票经纪人开发一款程序，用来记录他跟客户的交易记录。

你最奇怪的工作经历是什么？

“911”事件发生的时候，我恰好是 Delta 航空公司的执行财务。之前我在波士顿咨询工作，突然接到电话说要去 Delta 航空公司做财务工作，当时自己对这方面业务完全不懂。这段经历教会我，说“我不会”并不丢人，不懂装懂才可怕。

原文

http://www.kuailiyu.com/article/6348.html?utm_source=Tuicool_Weekly

程序员的敌人

天北京的天气简直好到爆，一整天下来，PM2.5 的值就没超过 25，俗话说好女不过百，好天不过 25，就是这个理儿。吃完晚饭，出去溜了一圈，天高云淡，月朗星稀，空气中弥漫着初冬的寒意，远处的灯光闪闪烁烁，透过清亮的空气，似乎可以看到一丝丝的光线发散出来，感觉真好。北京现在没什么土，也没什么沙，估计很多年前的沙尘暴是难得一见了，所以我希望，这个冬天，让大风来的更猛烈些……

MacTalk 的读者中一定有不少程序员，今天的提问环节是：程序员的敌人是谁？

好吧，举手比较踊跃，看来平时被虐的不轻啊，这位同学： 「Mac 君，我觉得是项目经理，我看丫那个得瑟劲儿就想抽……」 「坐下吧，一会走的时候把怀里的板砖留下。觉得是产品经理的也可以歇歇了」

「Mac 君，我脚着吧，这事和人无关，应该是 bug 搞的鬼，上线时节 bug 纷纷，长使英雄泪满襟！」 「这个答案虽然有了点人文关怀，不过 bug 不都程

序员写出来的吗，你不管谁管？」

插播：江湖传言，软件的不幸在于它创造出来的麻烦多过于它解决的麻烦。真实的情况是，软件只解决了一部分它创造出来的麻烦……

现在公布答案，程序员的敌人就是，当当当当，「需求」！

多少英雄汉，在程序世界里纵横四海独孤求败，结果被真实的需求碰球的鼻青脸肿头破血流黯然解甲，转售前的转售前，变甲方的变甲方，最可恨的是这些变成甲方的程序员，成了甲方之后开始用「需求」折磨其他的程序员，比甲方还甲方！原本同根生，相煎何太急！真特么像那只没抢到香蕉的猴子啊。

无数软件工程宝典、创业圣经和成功经验，都告诉我们，要想成功，挥刀自宫！哦，不对，要想成功，一定要满足用户的需求，做客户真正需要的东西。可是，你知道我们程序员想做到这一点有多难么？

首先，需求就分为了用户想要的（Want）和用户需要的（Need），其次，需求又分为企业用户需求和个人用户需求，第三，没有第三了。前两条就够我们程序员忙活一生了。

先说说 Want 和 Need，能够从 Want 直接转化成 Need 的需求简直凤毛麟角，你要遇到了就是你上辈子修来的福分，不当程序员天理不容。大部分情况下，Want 都是用户的谎言，他们不停的重复这些谎言，直到自己信以为真，然后告诉你，「兄弟，这就是哥的 Need，大胆的做吧，哥哥在岸上等着你」。

等你费劲扒拉把这个 A 做出来之后，客户拿在手中把玩一番之后，会满脸歉意的告诉你，「不好意思啊，兄弟，我要的是 B，你给我做出个 A 来干什么？」。你呕血三升之后，客户终于把 B 描述出来了，告诉你，做，这次一定不会错！但是你永远不知道后面还有 C、D、E、F、G……在等着……

这是从外部客户角度，还有苦逼的内部角度。比如微软的演讲奇才狮吼功大师鲍尔默，他老先生在 Surface 发布时宣称 「……但购买这几款平板电脑（iPad、Kindle 等）的用户有可能犯了一个错误，这并不是他们想要的产品。」「我不认为能有人像我这样意识到用户真正想要的是什么产品。你可以看看周围人们在用的那些平板电脑产品，没有一个是你真正能用的。苹果不行，谷歌不行，亚马逊也不行。没有一件产品可以既可以工作用又可以娱乐用……」「Surface 才是用户真正需要的产品……」

不知道说这些话的时候老先生在用哪个脚趾头思考，我特么怎么不知道范了那么多错误？Surface 的销量给了老鲍一记响亮的耳光，老鲍不干了。可怜那些研发 Surface 的程序员……

对于企业客户而言，几乎任何时候都存在 Want 和 Need 不对等的情况，我现在唯一能想到的应对之法就是「贴身服务」。再也不要打个包一百万让我们做 ABCDEFG 了，骗鬼呢？！单人单月报价，我们提供人员、技术、平台、工具、项目管理，您说吧，想做锤子还是锤子手机？按月付费，我陪您玩到天荒地老……

对于个人用户而言，如果你真的做对了，满足了用户的真正需求，可能有三种情况： 1、你是自己产品的真正用户 2、你运气不错 3、你是乔布斯

还有一点就是，为个人用户做产品不要让他们有太多选择，这些人不是程序员，不需要知道配置信息可以保存在 xml、注解、properties、json 中，这么多选项会让大家一起疯掉，给一个按钮就好了，就像 iPhone 那样。

Android 系统给了用户无限可能，但也被用户和开发者骂的狗血喷头。iOS 给了用户一堆限制和一个按钮，而且不能用 Flash，结果一个月以后，用户说：哦，原来我真的不需要 Flash！

「需求」如此残酷，值得为之奋斗！希望所有的程序员都能够做出真正满足客户需求的程序，十年后，我们的代码依然奔跑在千万台机器中，无数双手，在代码前挥舞…

原文 http://macshuo.com/?p=947&utm_source=Tuicool_Weekly

技术人创业可能面临的挑战

这儿我将从事技术工种的人员笼统地成为技术人，比如编辑，其实也是属于技术工种。引起我思考这个话题的是昨天公司年末的恳谈会，发现自己创业那么久，其实还是有很多挑战没有解决，比如控制欲和沟通等。

技术人，尤其是从事过软件研发的同学，比较明显的两个特点就是控制欲比较强，以及相对于面对面的沟通更偏向埋头工作。而这两点在创业过程中如果没有很好地平衡，可能会给自己和团队带来比较大的困扰。这儿还是分别以 InfoQ 团队发展过程中我们所犯的错误来举例说明，供大家参考。

少一些控制欲和条条框框，多一些信任

我也从事过软件研发工作，暂且不说工作做得怎么样（Ctrl+C，Ctrl+V 用的很熟），但是在几年的职业生涯中对编程也有基本的了解，后来从事技术编辑工作，和文字打交道。不论是代码还是文字，其实都是在自己的可控制范围之内的，你想修改成什么样子，就可以按照自己的思路去做，然后测试，部署等，一切按部就班。其实这就是一种典型的控制。

但和人打交道，尤其是和知识型工作者打交道时，如果还延续这种思维方式，就不是很合适。作为一个技术媒体，InfoQ 是一个典型的知识型工作者所组成的团队，大家智商和情商都比较高，其需求也比较明显，那就是相对自由的环境和决策，都比较强调自主性。一开始我并不是很了解这一点，鉴于自己是军校出身，多少更加强调纪律和规定，所以经常会定一些条条框框，防止大家出错，希望一切能够按照自己的设想前进。这其实也没有错，团队总是需要一个拿主意的人，但是事实来看，是有更好的方法的。

少一些条条框框，只有几个高压线，多一些指导和案例参考，让团队成员自己去拿主意、做决定，效果会更好。其原因也很简单，每个人都是主人。我们前一段时间试行了财务审批规定，很有意思，大家也都很自觉，按流程来走，造成的结果就是事事请示，大小金额均问是否可行，不然财务不批准。其实到目前来看，大家所提交的审批都是经过讨论和思考的，只是走一下流程，因为预算早就定了的或者必须要做的事情。如果拿掉审批，只采取报备，谁做了决定就告知大家一下，强调每个人的决策权和责任制，既加快了做事的速度，又提高了团队的信任感，何乐而不为？要知道，对于创业型团队，速度和信任都是宝贝，都是释放生产力的好办法。

灵魂人物多一些沟通比埋头苦干更好

在这次 InfoQ 年末的恳谈会上，不止一个同学提到，好像我与大家的距离远了，这是让我比较惊愕的事情。怎么想也不至于啊，不到 20 人的团队，怎么也不会出现这种情况啊。仔细了解过去，事实就是如此，因为过去的一年，团队打磨的事情比较耗精力，和大家 1V1 沟通的次数自然不自然地就少了很多。

早上阅读时正好也看到一个文章，提到某创业公司的离职率陡增，HR 和离职人员沟通后，有多人反馈“领导对公司和员工的重视程度有所减弱”。该公司的 CEO 赶紧反省，发现过去半年自己由于外务繁忙，将此前一直进行的周例会终止了几个月，然后员工就认为领导将心思和精力用在了公司之外的事情上，对公司的发展前景担忧。周例会正常之后，大家的情绪和干劲就又回到正常的轨道。

对于技术创业者来说，有时候为了让产品更加完美，或者项目执行的更好，更愿意埋头于实际的工作中，认为沟通都是比较虚的事情，画大饼是浪费时间的事情。当然，饼画的太多太大，而且没有实现，是件很悲催的事情，后果更加严重，因为基本的诚信失去了。但没有了“饼”，团队没有了发展的目标和前景，让大家所做的事情失去了“意义”，大家的精气神也会受到很大的影响，即使公司发展的还不错，团队成员的成就感和进取心也差许多。

所以，领头人平时有意识地将自己从埋头工作中走出来，多一些走动式的管理，和大家聊聊天。再加上一些例会，给大家一个提问的机会，了解公司发展状况的机会，将自己对于公司发展的设想掏心掏肺地告诉大家，也听听大家的反馈，其结果是让团队成员找到自己所做事情的意义，而自己的压力也会相对减轻，自己的思路有时候在大家的反馈下也能得到及时的纠正。

提醒自己智商和情商都不能忽视

简而言之，技术人的智商绝对没有问题，但情商有时候需要加强，像信任、沟通

等软技能更偏情商多一些。解决的办法一方面是自己多提醒自己，通过一些强制性的制度让自己少一些控制欲，多一些沟通。另外找个好的 Partner 也是比较重要的，假设扎克伯格没有其 COO 桑德伯格的支持，Facebook 的发展之路估计也不会那么顺利。

原文 http://taiwen.lofter.com/post/664ff_b2dc99?utm_source=Tuicool_Weekly